Success with C++
Kris Jamsa
Copyright © 1994 *Jamsa Press*

Succes cu C++
Kris Jamsa
Traducere: Ion Fătu

ISBN 973-9229-36-0

Editura **ALL EDUCATIONAL S.A.** Bd. Timişoara nr. 58, sector 6, cod 76548
☎ 312.11.46, 312.43.21, 311.07.44
Fax: 311.05.65

Departamentul difuzare: ☎ 312.18.21, 311.15.47

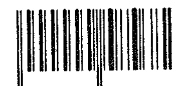
Redactori: *prof. Viorica Fătu și Mihai Mănăstireanu*
Tehnoredactare computerizată: *Roxana Cornelia Nistor*

PRINTED IN ROMANIA

SUCCES CU C++

KRIS JAMSA

Traducere: Ion Fătu





În colecția *software*

a editurii **ALL EDUCATIONAL**

Au apărut:

1. Microsoft Access pentru Windows 95, Ghid de referință - *James E. Powel*
© Sybex

2. Ghidul secret al calculatorului - *Russ Walter*
© Sybex

3. Word pentru Windows 95 - ușor și rapid - *Christian Crumlish*
© Sybex

4. Excel pentru Windows 95 - ușor și rapid - *Gerald E. Jones*
© Sybex

5. Microsoft Excel pentru Windows 95. Ghid de referință - *Douglas E. Hergert*
© Sybex

6. Primii pași în Internet - *Christian Crumlish*
© Sybex

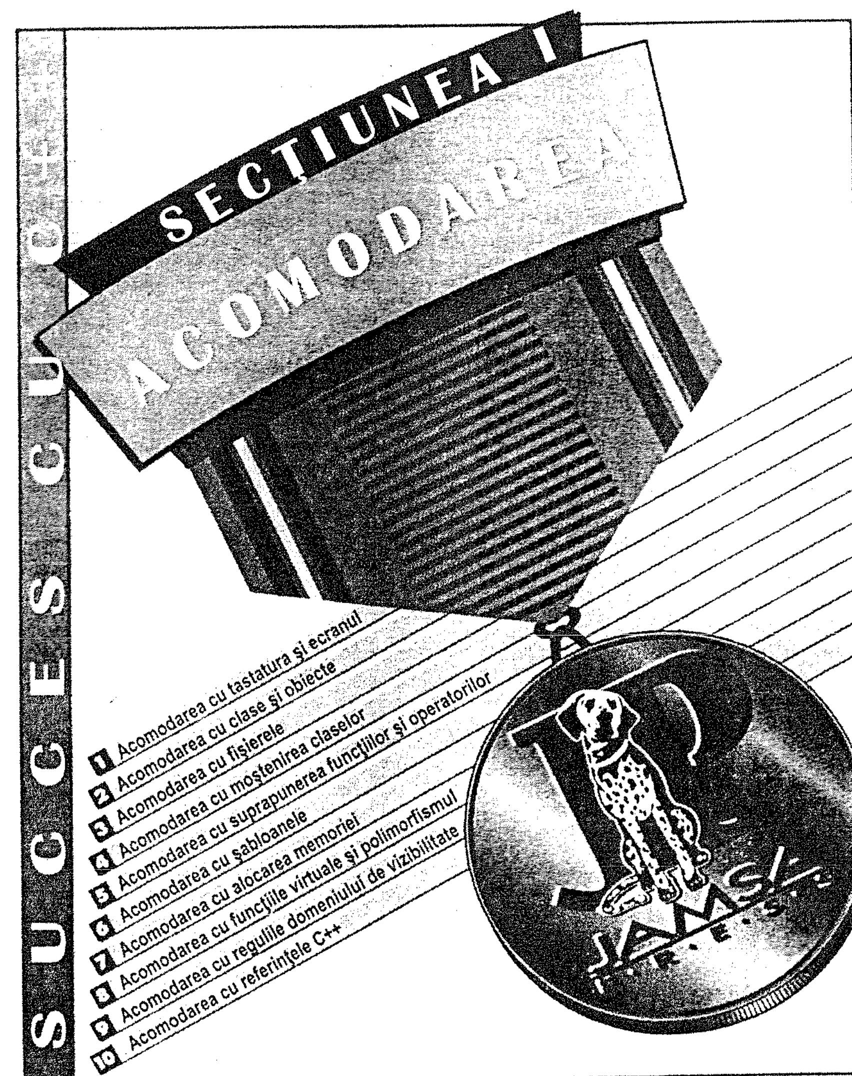
Vor apărea:

1. Biblioteca programatorului ActiveX - *Suleiman Lalani, Ramesh Chandak*
© Jamsa Press

2. Ghidul programatorului în Web - *Kris Jamsa, Suleiman Lalani, Steve Weakley*
© Jamsa Press

3. Biblioteca programatorului Java - *Suleiman Lalani, Kris Jamsa*
© Jamsa Press

4. Programarea în Internet - *Kris Jamsa, Ken Cope*
© Jamsa Press



CUPRINS

Secțiunea 1 - Acomodarea

Capitolul 1

Capitolul 10

Secțiunea 2: Aprofundarea

Capitolul 11

Capitolul 12

Capitolul 13

Capitolul 14

Capitolul 15

CAPITOLUL 1
ACOMODAREA CU TASTATURA ŞI ECRANUL

Fiecare program pe care îl creaţi, indiferent de scopul său, va conţine operaţii de intrare/ieşire. Acest capitol analizează în detaliu intrarea de la tastatură şi ieşirea pe ecranul monitorului. După terminarea acestui capitol, veţi cunoaşte:

- Ce este un stream I/O
- Cum foloseşte compilatorul un fişier antet
- De ce sunt importante *cin* şi *cout*
- Cum se poate formata intrarea şi ieşirea
- Ce este un manipulator
- Câteva utilizări ale funcţiilor componente ale stream-urilor de intrare şi ieşire
- Cum se realizează o ieşire prin buffer (memorie tampon)

Dacă sunteţi familiarizat cu operaţiile I/O din C++, se recomandă să revedeţi, totuşi, manipulatorii prezentaţi în acest capitol. Capitolul 11 examinează clasele I/O în detaliu, iar până atunci veţi face deja cunoştinţă cu clasele şi metodele I/O. Dacă sunteţi începători în C++, experimentaţi fiecare din programele prezentate aici; modificaţi-le şi observaţi rezultatele obţinute.

SĂ ÎNŢELEGEM STREAM-URILE I/O

În sensul cel mai simplu, un stream I/O este o secvenţă de caractere afişate pe ecranul monitorului sau citite de la tastatură. Pentru operaţiile standard de intrare/ieşire, în C++ se folosesc stream-urile **cin** (pentru intrare) şi **cout** (pentru ieşire). După cum se va vedea, C++ defineşte stream-urile I/O prin clase. Definiţia acestora apare în fişierele antet IOSTREAM.H sau ISTREAM.H sau OSTREAM.H, în funcţie de compilatorul pe care îl folosiţi. Dacă examinaţi conţinutul acestor fişiere, veţi vedea declaraţiile pentru diferite metode şi operatori de intrare/ieşire. Este chiar indicat să tipăriţi conţinutul acestor fişiere pentru a-l folosi ca referinţă. Nu fiţi îngrijoraţi dacă nu înţelegeţi toate declaraţiile şi definiţiile ce apar în aceste fişiere; le vom examina în detaliu în Capitolul 11.

Observaţie: *Drept exemplu de diferenţe între compilatoare, Borland C++ foloseşte IOSTREAM.H pentru majoritatea definiţiilor legate de stream-uri, pe când compilatorul Microsoft Visual C++ foloseşte pentru aceste definiţii fişierele IOS.H, ISTREAM.H, OSTREAM.H şi STREAMB.H.*

De obicei, programul C++ va folosi fişierul IOSTREAM.H drept una din primele instrucţiuni. Următorul program, SUCCESS.CPP, foloseşte **cout** pentru a afişa un mesaj pe ecran:

```
#include <iostream.h>

void main(void)
 {
   cout << "Getting up to speed with C++ I/O";
 }
```

După cum se vede, programul foloseşte operatorul de ieşire sau *de inserţie* (semnul <<) pentru a afişa mesaje pe ecran. În capitolul 11 veţi învăţa modul de definire a operatorului de inserţie C++pentru clasele de stream-uri I/O. Tot în acel capitol veţi descoperi cum se defineşte **cout** ca un obiect.

Următorul program, GET_NAME.CPP, cere utilizatorului să-şi introducă prenumele, apoi foloseşte operatorul de intrare sau de *extragere* (semnul >>) pentru a atribui variabilei *first_name* literele introduse :

```
#include <iostream.h>

void main(void)
 {
   char first_name[64];

   cout << "Type in your first name: ";

   cin >> first_name;

   cout << "Hello, " << first_name;
 }
```

Programul foloseşte operatorul de ieşire pentru a afişa solicitarea de introducere a numelui utilizatorului, după care, folosind operatorul de intrare, atribuie variabilei *first_name* literele tastate. Apoi, foloseşte încă o dată operatorul de ieşire pentru a afişa mesajul „Hello", urmat de numele introdus:

```
cout << "Hello, " << first_name;
```

Când tipăriţi informaţii folosind operatorii de extracţie multiplă în acest mod, C++ afişează informaţia aşa cum apare în instrucţiune, de la stânga spre dreapta. Rulaţi programul anterior şi introduceţi prenumele, urmat apoi de numele dvs. Veţi vedea că programul afişează numai prenumele, indiferent de informaţia introdusă. Când folosiţi *cin* pentru a citi şiruri de caractere (cum ar fi un nume) de la tastatură, cin foloseşte spaţiile albe (spaţiu, sau tab, sau sfârşit de linie) pentru a separa şirurile. Dacă aveţi nevoie să citiţi o linie de text de la tastatură, puteţi folosi funcţia membru *cin.getline*, ce va fi prezentată puţin mai târziu. Următorul program, FULLNAME.CPP, plasează două variabile în instrucţiunea de intrare pentru a citi atât prenumele, cât şi numele utilizatorului:


```
#include <iostream.h>

void main(void)
 {
   char first_name[64];
   char last_name[64];

   cout << "Type your first and last names: ";
   cin >> first_name >> last_name;

   cout << "Hello, " << first_name << " " << last_name;
 }
```

Programul foloseşte următoarea instrucţiune pentru a citi valorile la două variabile:

```
cin >> first_name >> last_name;
```

Când folosiţi doi sau mai mulţi operatori de extragere în aceeaşi instrucţiune, C++ atribuie variabilelor valorile de la intrare, începând de la stânga spre dreapta. Observaţi, de asemenea, folosirea ghilimelelor în mesajul de ieşire, pentru a separa prenumele de nume. Dacă îndepărtaţi ghilimelele, numele va fi scris, fără pauză, în continuarea prenumelui.

```
cout << "Hello, " << first_name << " " << last_name;
```



SĂ ÎNŢELEGEM CIN ŞI COUT

Un stream I/O este o secvenţă de caractere, pentru ieşire sau pentru intrare. Când un program trebuie să afişeze rezultate pe ecranul monitorului (sau pe dispozitivul standard de ieşire), acesta trebuie să insereze caracterele în stream-ul de ieşire **cout** folosind operatorul de inserţie astfel:

```
cout << "Success with C++" << endl;
```

Când un program primeşte intrare de la tastatură (sau de la un dispozitiv standard de intrare), acesta extrage caractere din stream-ul de intrare **cin**, folosind operatorul de extragere sub forma:

```
cin << nume_variabila;
```

REDIRECTAREA STREAM-URILOR I/O

C++ asociază stream-urile *cin* şi *cout* cu dispozitive standard de intrare şi ieşire ale sistemului de operare. În acest mod, se pot folosi stream-urile I/O pentru a scrie programe ce redirijează intrarea şi ieşirea. De exemplu, următoarea linie de comandă cere sistemului DOS (sau UNIX) să trimită ieşirea programului SUCCESS spre un fişier numit OUTPUT.DAT, şi nu pe ecranul monitorului:

```
C:\> SUCCESS > OUTPUT.DAT <ENTER>
```

Următorul program, LINE_NBR.CPP, afişează un număr în faţa fiecărei linii a intrării redirijate.

```
#include <iostream.h>

void main(void)
 {
   char line[256];
   long line_number = 0;

   while (! cin.eof())                    // Loop until no characters
                                          // are available
     {
       cout << ++line_number << '\t';     // Display a line number
                                          // and a tab
       cin.getline(line, sizeof(line));   // Read a line of text
       cout << line << endl;              // Display the line of text
     }
 }
```

Programul foloseşte stream-urile de intrare/ieşire *cin* şi *cout*. Totuşi, aşa cum se observă, acesta beneficiază de existenţa a două funcţii membru, *cin.eof* şi *cin.getline*. Să ne reamintim că *cin* şi *cout* sunt obiecte, deci definesc funcţii membre. Dacă n-aţi folosit niciodată aceste două funcţii nu vă faceţi griji. Le vom prezenta mai târziu, în această lecţie. Pe lângă folosirea acestor două funcţii, programul foloseşte manipulatorul *endl* pentru a scrie caracterul newline (linie nouă), compus din caracterele carriage-return (retur de car) şi line-feed (avans hârtie).

Pentru a afişa numărul fiecărei linii a fişierului SUCCESS.CPP, de exemplu, se poate utiliza LINE_NBR astfel:

```
C:\> LINE_NBR  <  SUCCESS.CPP  <ENTER>
1       #include <iostream.h>
2
3       void main(void)
4        {
5          cout << "Getting up to speed with C++ I/O";
6        }
```


Dacă se doreşte salvarea rezultatelor pe un fişier (de exemplu, pe fişierul OUTPUT.DAT), se poate redirija ieşirea programului LINE_NBR astfel:

```
C:\>LINE_NBR <SUCCESS.CPP> OUTPUT.DAT <Enter>
```

FOLOSIREA ALTOR STREAM-URI I/O

Majoritatea programelor C++ pe care le veţi întâlni folosesc stream-urile de I/O *cin* şi *cout*. Aşa cum se va vedea, fişierul IOSTREAM.H defineşte şi stream-urile I/O *cerr* şi *clog*. Tabela 1.1. descrie pe scurt diferite stream-uri I/O:

Stream	Scop	Exemplu
cin	intrare tastatură (stdin)	cin >> name;
cout	ieşire ecran (stdout)	cout <<"Hello, world!";
cerr	ieşire dispozitiv erori standard (stderr)	cerr <<"Eroare critică..;
clog	ieşire buffer-izată pe dispozitiv erori standard (stderr)	clog <<"Mesaj eroare..;

Tabela 1.1. Rezumat al stream-urilor I/O din C++

REDIRIJAREA IEŞIRII PROGRAMELOR

Când se rulează un program, C++ asociază stream-urile I/O *cin* şi *cout* cu dispozitivele standard de intrare, respectiv ieşire. În mod prestabilit, sistemul de operare asimilează dispozitivul standard de intrare cu tastatura, iar dispozitivul standard de ieşire cu ecranul monitorului. Dacă se apelează un program C++ folosind operatorii de redirijare I/O în linia de comandă, se poate redirija intrarea şi ieşirea programului. Cu alte cuvinte, se poate redirija stream-ul de ieşire *cout* de la ecran spre un fişier sau un alt program. În mod asemănător, se poate redirija stream-ul de intrare *cin* de la tastatură spre un fişier.

Când un program caută şi procesează erorile critice, mesajele de eroare nu pot fi redirijate de la ecranul monitorului. În acest caz trebuie folosit stream-ul **cerr** pentru afişarea mesajelor. De exemplu, următorul program, USE_CERR_CPP afişează un mesaj folosind *cerr*:

```
#include <iostream.h>

void main(void)
 {
   cerr << "You cannot redirect this message";
 }
```

Încercaţi să compilaţi şi să executaţi acest program, folosind operatorii de redirijare ai sistemului de operare. Veţi vedea că nu este posibilă redirijarea

mesajelor scrise cu *cerr*. Mai târziu, tot în acest capitol, veţi examina ieşirea prin buffer folosită de *cout* şi *clog*.



SĂ ÎNŢELEGEM STREAM-URILE I/O

În cel mai simplu sens, un stream I/O este o succesiune de caractere, scrise pe ecran sau citite de la tastatură. Fişierul antet IOSTREAM.H defineşte patru stream-uri I/O: *cout*, *cin*, *cerr* şi *clog*. Stream-ul *cout* permite afişarea rezultatelor pe ecran sau pe dispozitivul de ieşire standard. În mod asemănător, *cin* realizează citirea caracterelor de la tastatură sau de la dispozitivul standard de intrare. În sfârşit, *cerr* şi *clog* determină programele să afişeze ieşirea pe dispozitivul de eroare standard. Pentru a transmite ieşirea spre un stream I/O, se foloseşte operatorul de extragere (<<), ca mai jos:

```
cout <<"Hello, successful world!“;
cerr <<Mesaj de eroare“;
```

Pentru a obţine intrare de la *cin*, se foloseşte operatorul de inserţie (>>), ca mai jos:

```
cin >> nume;
```

C++ permite scrierea mai multor operatori sau valori de date în stream-ul de ieşire sau a unor variabile în stream-ul de intrare:

```
cont <<"introduceti numele si prenumele“<< endl;
cin >> nume>> prenume;
```

SĂ ÎNŢELEGEM STREAM-URILE DE INTRARE ŞI IEŞIRE

Mulţi dintre manipulatorii şi funcţiile discutate în acest capitol corespund stream-urilor de intrare şi ieşire. La examinarea fişierului IOSTREAM.H veţi găsi unele definiţii ce folosesc *ios* şi altele ce folosesc *istream* şi *ostream*. Definiţiile ce folosesc *ios* corespund stream-urilor intrare şi ieşire (de aceea apare prescurtarea io). În mod similar, definiţiile bazate pe *istream* corespund stream-urilor de intrare, iar cele bazate pe *ostream* - stream-urile de ieşire. Prin examinarea definiţiilor stream-urilor se poate înţelege mai bine fiecare manipulator sau funcţie membru. În Capitolul 11 se vor trata mai detaliat relaţiile între clase.

CARACTERE SPECIALE

Unul din programele precedente a folosit manipulatorul *endl* pentru a genera o secvenţă CR/LF (retur de car - avans hârtie). În mod asemănător, următorul program, THREELIN.CPP, foloseşte un manipulator *endl* pentru a afişa ieşirea pe trei linii.


```
#include <iostream.h>

void main(void)
 {
   cout << "Success„ << endl;
   cout << "with„ << endl;
   cout << "C++!„ << endl;
 }
```

Manipulatorul *endl* se poate folosi de mai multe ori pe aceeaşi linie, ca în programul ce urmează, THREETOO.CPP, ce este funcţional identic cu cel precedent:

```
#include <iostream.h>

void main(void)
 {
   cout << "Success" << endl << "with" << endl << "C++!" << endl;
 }
```

În mod similar, un alt program prezentat anterior în acest capitol folosea caracterele \t pentru a genera tabulatori. Când se realizează ieşirea pe *cout*, *cerr* sau *clog* în stream-ul de ieşire se pot folosi caracterele speciale prezentate în Tabela 1.2.

Caracterul	Semnificaţia
\a	Alertă sau clopoţel
\b	Backspace
\f	Început de pagină
\n	Început de linie nouă (echivalent cu *endl*)
\r	Carriage return (începutul liniei curente) fără avans hârtie (linefeed)
\t	Tabulator orizontal
\v	Tabulator vertical
\\	Backslash
\?	Semn de întrebare
\'	Apostrof
\"	Ghilimele
\0	Caracter nul
\ooo	Valoare în octal, ca \033
\xhhh	Valoare hexazecimală, ca \x1B

Tabela 1.2. Caractere speciale folosite cu **cout, cerr** *şi* **clog**

Următorul program, SPECCHAR.CPP, foloseşte aceste simboluri speciale pentru a şterge ecranul şi apoi pentru a emite un semnal sonor prin difuzorul calculatorului. Pentru ştergerea ecranului, trebuie să fie încărcat driver-ul de dispozitiv ANSI. Programul foloseşte secvenţa specială ANSI **Esc[2J** pentru a şterge ecranul:

```
#include <iostream.h>

void main(void)
 {
   // Clear the screen display
   cout << "\033[2J";

   // Sound the computer's speaker
   cout << "Beep\a\tBeep\a\t\Beep\a";
 }
```

Programul foloseşte caracterul \a pentru semnalul sonor şi caracterul \t pentru avansul cursorului cu următorul spaţiu de tabulare. Multe din programele prezentate în această carte vor folosi diferite caractere speciale.

Caracterele speciale pot fi folosite cu fiecare din stream-urile de ieşire, la fel şi cu stream-urile de fişier din Capitolul 3. De exemplu, următorul program, BEEPCERR.CPP, foloseşte caracterul \a pentru emiterea semnalului sonor înainte de afişarea unui mesaj către *cerr*:

```
#include <iostream.h>

void main(void)
 {
   cerr << "\aSome error message\t\tXXX-11-XXXX";
 }
```

După cum se poate vedea, programul foloseşte caracterul \a pentru semnalul sonor şi caracterul \t pentru a genera un tabulator. În Capitolul 11 veţi învăţa cum să creaţi propriii dumneavoastră manipulatori ce corespund acestor caractere speciale.

LUCRUL CU MANIPULATORI

Un *manipulator* de stream I/O este un element prin care se filtrează ieşirea unui stream I/O. De exemplu, manipulatorii *hex, oct* şi *dec* determină afişarea unei valori în hexazecimal, octal sau zecimal. Următorul program, MANIPULA.CPP, ilustrează modul în care sunt folosiţi aceşti manipulatori.

```
#include <iostream.h>

void main(void)
```



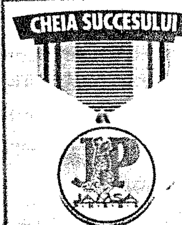
```
  int number = 1001;

  cout << "Decimal: " << number << "\tHexadecimal: " << hex <<
          number << endl;
  cout << "Decimal: " << dec << number << "\tOctal: " << oct <<
          number << endl;
}
```

Dacă examinaţi a treia instrucţiune a programului, veţi observa folosirea manipulatorului *dec*. Ar putea părea surprinzător de ce este necesar să folosim manipulatorul cu valoarea zecimală 1001. Aşa cum se va vedea, stream-ul de ieşire *cout* afişează valorile, în mod prestabilit, în zecimal. Dacă programul foloseşte manipulatorii *oct* sau *hex, cout* va afişa valorile în octal, respectiv în hexazecimal, în funcţie de ultimul manipulator folosit. De aceea, pentru a tipări valoarea 1001 în zecimal, programul trebuie să folosească manipulatorul *dec*. Experimentaţi programul, adăugând eventual următoarea instrucţiune la sfârşitul acestuia:

```
cout << "Afisarea valorii 1001:" << 1001;
```

Deoarece manipulatorul *oct* a fost ultimul folosit în program, *cout* va afişa valoarea în octal, nu în zecimal, aşa cum poate părea la prima vedere.



STABILIREA BAZEI DE CONVERSIE

Manipulatorii *dec, hex* şi *oct* ajută la stabilirea bazei de conversie pe care o va folosi un stream de ieşire pentru afişarea numerelor. De exemplu, următoarea instrucţiune dirijează stream-ul *cout* să tipărească valoarea 255 în hexazecimal:

```
cout << hex << 255;
```

După execuţia instrucţiunii, totuşi, stream-ul *cout* va folosi baza hexazecimală pentru toate ieşirile de tip numeric, până se va selecta o nouă bază. Nu este obligatoriu să tipăriţi o valoare pentru a schimba baza. Următoarea instrucţiune, de exemplu, dirijează *cout* pentru a afişa ulterior valori în zecimal:

```
cout <<dec;
```

În sfârşit, când se specifică o bază folosind manipulatorii *dec, hex* sau *oct*, această bază va fi specifică unui stream I/O. Bazele celorlalte stream-uri nu vor fi afectate.

Pe lângă manipulatorii *dex, hex* şi *oct* de stabilire a bazei de conversie, într-un program se poate folosi şi manipulatorul *setbase* care este definit în fişierul header IOMANIP.H. Următorul program, SETBASE.CPP, foloseşte manipulatorul *setbase* pentru a obţine acelaşi rezultat ca şi programul anterior:

```
#include <iostream.h>
#include <iomanip.h>

void main(void){
  int number = 1001;

  cout << "Decimal: " << number << "\tHexadecimal: " <<
          setbase(16) << number << endl;
  cout << "Decimal: " << setbase(10) << number << "\tOctal: " <<
          setbase(8) << number << endl;
}
```

***Observaţie:** Dacă compilatorul pe care îl folosiţi nu include manipulatorul* **setbase**, *puteţi folosi manipulatorul* **setiosflags**. *De exemplu, pentru a selecta baza hexazecimală, folosiţi* **setiosflags (ios::hex)**.

CONTROLUL LĂŢIMII ZONEI DE ALINIERE

Câteva din programele anterioare au folosit spaţii în interiorul şirurilor de caractere pentru a formata afişarea acestora. Pe lângă această metodă „brutală" de spaţiere, programele pot folosi manipulatorul *setw* care specifică numărul minim de poziţii ale zonei de alinere. Dacă o valoare necesită mai multe caractere decât numărul specificat în *setw*, atunci pentru afişarea ei vor fi folosite atâtea caractere câte sunt necesare. Dacă o valoare necesită mai puţine caractere, atunci spaţii suplimentare vor fi inserate înaintea valorii afişate. Spre deosebire de manipulatorii *dec, oct, hex* şi *setbase* prezentaţi mai înainte, manipulatorul *setw* are efect numai asupra valorii următoare. Cu alte cuvinte, alinierea stabilită prin *setw* nu este permanentă. Următorul program, SETW.CPP, ilustrează folosirea manipulatorului *setw*:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  int i;

  for (i = 1; i < 5; i++)
   cout << setw(i) << 1 << setw(i) << 12 << setw(i) << 123 << endl;
 }
```

Deoarece lăţimea zonei de aliniere selectată prin *setw* influenţează numai următoarea valoare afişată, programul foloseşte manipulatorul de câteva ori în aceeaşi instrucţiune. La compilarea şi execuţia acestui program, pe ecran vor fi afişate următoarele:

```
C:\> SETW  <ENTER>
112123 ———————————— setw(1)
 112123 ——————————— setw(2)
```



```
 1 12123 ——————————— setw(3)
  1  12 123 ———————— setw(4)
```

După cum se poate observa, dacă o valoare are mai multe caractere decât cele selectate de *setw*, ea va fi afişată cu toate caracterele necesare. În cazul primei linii de ieşire, toate cele trei valori conţin cel puţin o cifră. În a doua linie sunt alocate două caractere pentru fiecare valoare, în a treia linie - trei, iar în a patra linie - 4 caractere. Se observă că în ultima linie, valoarea 1 este precedată de 3 spaţii, valoarea 12 - de două spaţii, iar valoarea 123 - de un spaţiu.

STABILIREA CARACTERULUI DE UMPLERE

Aşa cum aţi învăţat, manipulatorul *setw* permite specificarea lăţimii minime a zonei de aliniere. În mod prestabilit, stream-urile I/O folosesc spaţiile pentru obţinerea unei spaţieri corecte între caractere. Cu ajutorul manipulatorului *setfill* se pot specifica şi alte caractere de completare a zonelor dintre valori. De exemplu, următoarea instrucţiune dirijează stream-ul *cout* pentru a folosi punctul drept caracter de completare:

```
cout <<setfil ('.');
```

Când se specifică un caracter prin *setfill*, acel caracter rămâne în vigoare pentru toate afişările ulterioare, până când se selectează un alt caracter. Următorul program, SETFILL.CPP, schimbă programul precedent SETW.CPP pentru a folosi caracterul '.' drept caracter de umplere:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  int i;

  cout << setfill('.');

  for (i = 1; i < 5; i++)
   cout << setw(i) << 1 << setw(i) << 12 << setw(i) << 123 << endl;
 }
```

Prototipul pentru manipulatorul *setfill* se găseşte în fişierul antet IOMANIP.H. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SETFILL  <ENTER>
112123
.112123
..1.12123
...1..12.123
```

O utilizare frecventă a manipulatorului *setfill* este la crearea unui meniu sau a unei table de cuprins, după cum se vede în continuare:

Cuprins
Capitolul 1. Iniţierea în intrări/ieşiri cu tastatura.................... 1
Capitolul 2. Iniţierea în clase.................................. 22
Capitolul 3. Iniţierea în fişiere................................ 45
Capitolul 4. Iniţierea în clase template 67
Capitolul 5. Iniţierea în moştenire.............................. 99

Următorul program, TABLE.CPP, ilustrează modul în care manipulatorul *setfilll* creează tabela precedentă:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << setfill('.');
   cout << "Table of Contents" << endl;
   cout << "Chapter 1 Getting Up to Speed with Keyboard I/O" <<
           setw(18) << 1 << endl;
   cout << "Chapter 2 Getting Up to Speed with Classes" <<
           setw(34) << 22 << endl;
   cout << "Chapter 3 Getting Up to Speed with Files" <<
           setw(36) << 45 << endl;
   cout << "Chapter 4 Getting Up to Speed with Templates" <<
           setw(32) << 67 << endl;
   cout << "Chapter 5 Getting Up to Speed with Inheritance" <<
           setw(30) << 99 << endl;
 }
```

CONTROLUL AFIŞĂRII VALORILOR ÎN VIRGULĂ MOBILĂ[1]

Când se lucrează cu numere zecimale, ca de exemplu 3,14159, se poate controla numărul cifrelor afişate prin folosirea manipulatorului *setprecision*. Să presupunem că dorim să tipărim rezultatul expresiei 22,0/7,0. Pentru a controla numărul de zecimale afişate se foloseşte manipulatorul *setprecision*, ca în programul SETPREC.CPP:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
```

[1] După cum se ştie, sistemul zecimal anglo-saxon foloseşte virgula în locul punctului şi invers. – N.R.


```
{
  int i;

  for (i = 0; i < 10; i++)
    cout << setprecision(i) << 22.0 / 7.0 << endl;
}
```

Aşa cum se vede, programul ciclează de la 0 la 10, folosind *setprecision* pentru controlul numărului de cifre afişate. Prototipul manipulatorului *setprecision* este definit în fişierul header IOMANIP.H. La compilare şi execuţia acestui program, pe ecran vor fi afişate următoarele:

```
C:\> SETPREC <ENTER>
3.142857
3.1
3.14
3.143
3.1429
3.14286
3.142857
3.1428571
3.14285714
```

Observaţii: *Tot în acest capitol veţi învăţa să selectaţi formatul fix de afişare a numerelor zecimale folosind indicatorul* ios::fixed. *La folosirea unui format fix, manipulatorul* setprecision *controlează numărul de zecimale afişate, şi nu numărul total de cifre.*

GOLIREA BUFFER-ULUI

Aşa cum aţi învăţat, stream-urile de ieşire cout şi clog folosesc *ieşirea prin buffer* (memoria tampon), adică datele sunt memorate într-o locaţie de memorie şi apoi transferate simultan, în întregime, pentru o mai mare eficienţă. Tot în acest capitol, veţi examina în detaliu ieşirea prin buffer şi ieşirea normală (fără buffer). Veţi învăţa că, atunci când realizaţi ieşirea prin buffer, sunt momente când doriţi ca programul să transfere imediat datele (să le mute din locaţia buffer-ului) pe ecran. Pentru a obţine ieşirea în acest mod, se poate folosi manipulatorul flush. Programele prezentate în secţiunea respectivă vor explica în detaliu noţiunile legate de manipulatorul flush (fără buffer).

IGNORAREA SPAŢIILOR ALBE CE PRECED DATELE DE INTRARE

După cum ştiţi, stream-ul de intrare *cin* foloseşte spaţii albe pentru a separa câmpurile de intrare. Când realizaţi operaţii I/O, există cazuri când *cin* trebuie să ignore spaţiile anterioare primului câmp de informaţii. Pentru aceasta se poate folosi manipulatorul *ws*, ca mai jos:

```
cin >> ws >> camp;
```

De exemplu, următorul program, SKIL_WS.CPP, foloseşte manipulatorul *ws* pentru ignorarea spaţiilor albe plasate înaintea textului tipărit:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   char text[128];

   cout << "Type in a word preceded by one or more blanks" << endl;
   cin >> ws >> text;
   cout << "You typed >" << text << "<" << endl;
 }
```

CONTROLUL INDICATORILOR STREAM-ULUI I/O

Aşa cum aţi învăţat, o parte din manipulatorii stream-ului I/O rămân activi şi după încheierea unei operaţii de intrare/ieşire. De exemplu, manipulatorii *dec*, *hex* şi *oct* selectează baza de afişare a valorilor numerice. Pentru a implementa aceşti manipulatori, stream-urile I/O au un câmp indicator (flag field) care specifică valorile curente ale parametrilor. Un fişier antet defineşte aceste valori ca fiind de tip enumerativ, după cum se vede mai jos:

```
enum {
  skipws =  0x0001,      // Skip white space during input
  left =  0x0002,        // Left justify output
  right = 0x0004,        // Right justify output
  internal = 0x0008,     // Pad after sign or base indicator
  dec =  0x0010,         // Decimal base
  oct =  0x0020,         // Octal base
  hex = 0x0040,          // Hexadecimal base
  showbase = 0x0080,     // Use base indicator on output
  showpoint = 0x0100,    // Force decimal point for floating point
  uppercase = 0x0200,    // Uppercase hex output
  showpos = 0x0400,      // Add '+' to positive integers
  scientific =  0x0800,  // Use 3.1415E2 floating notation
  fixed = 0x1000,        // Use 314.45 floating notation
  unitbuf = 0x2000,      // Flush all streams after insertion
  stdio = 0x4000         // Flush stdout, stderr after insertion
 }
```

Prin folosirea manipulatorilor *setiosflags*, programele pot controla o bună parte dintre aceşti indicatori. De exemplu, următorul program, IOSFLAGS.CPP, foloseşte indicatorii *dec, oct, hex* şi *showbase* pentru a afişa un număr în zecimal, octal şi hexazecimal:


```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << setiosflags(ios::showbase);

   cout << setiosflags(ios::dec);
   cout << "10 in decimal is " << 10 << endl;

   cout << setiosflags(ios::oct);
   cout << "10 in octal is " << 10 << endl;

   cout << setiosflags(ios::hex);
   cout << "10 in hex is " << 10 << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> IOSFLAGS <ENTER>
10 in decimal is 10
10 in octal is 012
10 in hex is 0xa
```

În acest caz, indicatorul *ios::showbase* dirijează stream-ul *cout* să prefixeze valorile octale cu 0, iar cele hexazecimale cu Ox. În unele cazuri, poate fi nevoie de setarea mai multor indicatori simultan. Aceasta se poate realiza printr-o operaţie OR (SAU) între indicatorii doriţi, cum se vede în continuare:

```
cout <<setiosflags (ios::showbase / ios::oct);
```

AFIŞAREA VALORILOR HEXAZECIMALE CU MAJUSCULE

După cum ştiţi, manipulatorul *hex* permite afişarea unei valori în hexazecimal. În mod prestabilit, stream-urile I/O afişează valorile hexazecimale cu litere mici. Pentru ca acestea să fie afişate cu majuscule, se poate folosi indicatorul *ios::uppercase*. Programul următor, UPPERCASE.CPP, prezintă modalitatea de utilizare a indicatorului menţionat.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << "Lowercase: " << hex << 255 << "  " << 10 << endl;
   cout << setiosflags(ios::uppercase);
   cout << "Uppercase: " << 255 << "  " << 10 << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> UPPERHEX <ENTER>
Lowercase: ff a
Uppercase: FF A
```

ALINIEREA REZULTATELOR LA STÂNGA SAU LA DREAPTA

În mod prestabilit, stream-urile I/O afişează ieşirea aliniată la stânga. Prin folosirea indicatorilor *ios::left* şi *ios::right*, programele pot controla alinierea ieşirii. De exemplu, următorul program, LEFTRITE.CPP, foloseşte indicatorii menţionaţi pentru a alinia ieşirea atât la stânga, cât şi la dreapta:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  cout << setiosflags(ios::right);
  cout << setw(5) << 1 << setw(5) << 2 << setw(5) << 3 << endl;
  cout << setiosflags(ios::left);
  cout << setw(5) << 1 << setw(5) << 2 << setw(5) << 3 << endl;
 }
```

După compilarea şi execuţia programului, ecranul va arăta astfel:

```
C:\> LEFTRITE <ENTER>
    1    2    3
1    2    3
```

După cum s-a văzut, programul anterior ar fi putut conţine instrucţiuni combinate, astfel:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  cout << setiosflags(ios::right) << setw(5) << 1 << setw(5) <<
         2 << setw(5) << 3 << endl;
  cout << setiosflags(ios::left) << setw(5) << 1 << setw(5) <<
         2 << setw(5) << 3 << endl;
 }
```

CONTROLUL AFIŞĂRII CU PUNCT FIX ŞI CU EXPONENT

Valorile reale pot fi afişate atât în format zecimal fix, ca de exemplu 123.456, cât şi în format cu exponent (notaţie ştiinţifică), cum ar fi 1.23456e2. Indicatorii *ios::fixed* şi *ios::scientific* permit controlul afişării numerelor reale. Următorul program, FIXFLOAT.CPP, ilustrează folosirea acestor doi indicatori de formatare:


```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  cout << setiosflags(ios::fixed) << 123.45 << endl;
  cout << 12345.6789 << endl;
  cout << resetiosflags(ios::fixed);
  cout << setiosflags(ios::scientific) << 123.45 << endl;
  cout << 12345.6789 << endl;
 }
```

Observaţie: *Programul foloseşte instrucţiunea* **resetios flags (ios::fixed)** *pentru preîntâmpinarea erorilor ce pot apărea la rularea programelor din această carte cu unele compilatoare. Testaţi acest program folosind compilatorul pe care îl aveţi la dispoziţie şi decideţi dacă puteţi elimina această instrucţiune.*

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> FIXFLOAT <ENTER>
123.450000
12345.678900
1.234500e+02
1.234568e+04
```

După cum se observă, formatul *ios::fixed* dirijează stream-ul I/O pentru afişarea punctului zecimal în poziţia sa actuală, în timp ce indicatorul *ios::scientific* dirijează stream-ul I/O să folosească formatul exponenţial.

FORŢAREA AFIŞĂRII PUNCTULUI ZECIMAL

Când programele afişează rezultatele unor operaţii în virgulă mobilă, nu întotdeauna punctul zecimal este tipărit, în special când rezultatele sunt exacte (fără parte zecimală). Să considerăm, de exemplu, următoarea instrucţiune:

```
cout <<10.0/5 << endl;
```

Deoarece rezultatul împărţirii este exact, stream-ul I/O poate tipări valoarea 2, în loc de 2.0. Dacă se doreşte şi tipărirea punctului zecimal, se poate folosi indicatorul *ios::showpoint*. Următorul program, SHOWPOIN.CPP ilustrează folosirea acestui indicator pentru forţarea afişării punctului zecimal.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  cout << 10.0 / 5 << endl;
  cout << setiosflags(ios::showpoint) << 10.0 / 5 << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SHOWPOIN  <ENTER>
2
2.00000
```

FORŢAREA AFIŞĂRII SEMNULUI UNEI VALORI

În mod prestabilit, stream-urile I/O afişează numai semnul valorilor negative. În funcţie de scopul programului, uneori este necesară afişarea semnului plus în faţa valorilor pozitive. În astfel de situaţii, programele pot folosi indicatorul *ios::showpos*. De exemplu, următorul program, SHOWPOS.CPP, ilustrează folosirea acestui indicator.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << -10 << " " << -5 << " " << 0 << " " << 5 << " " <<
          10 << endl;
   cout << setiosflags(ios::showpos);
   cout << -10 << " " << -5 << " " << 0 << " " << 5 << " " <<
          10 << endl;
 }
```

După compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SHOWPOS  <ENTER>
-10 -5 0 5 10
-10 -5 0 +5 +10
```

REFACEREA INDICATORILOR STREAM-URILOR I/O

În acest paragraf aţi învăţat cum se selectează anumiţi indicatori prin folosirea manipulatorului *setiosflag*. Dacă programele execută o mare cantitate de stream I/O, uneori este necesară activarea/dezactivarea indicatorilor. Pentru a readuce un indicator la valoarea sa prestabilită, programele pot folosi manipulatorul *resetiosflags*. De exemplu, urmǎtorul program, RESETIO.CPP, va afişa diferite valori în hexazecimal, precedate de specificatorul bazei hexazecimale Ox. Programul foloseşte apoi manipulatorul *resetiosflags* pentru a invalida afişarea specificatorului bazei:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
  cout << setiosflags(ios::showbase);
  cout << hex << 255 << " " << 10 << endl;
  cout << resetiosflags(ios::showbase);
  cout << 255 << " " << 10 << endl;
}
```


La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> RESETIO  <ENTER>
0xff 0xa
ff a
```



FOLOSIREA INDICATORILOR I/O

Manipulatorii stream-urilor permit programelor să controleze diferite setări I/O, prin păstrarea în stream-urile I/O a unei colecţii de biţi indicatori (flag bits). Prin folosirea manipulatori *setiosflags* şi *resetiosflags*, programele pot controla direct starea indicatorilor. De exemplu, următoarea instrucţiune dirijează stream-ul *cout* să afişeze informaţiile aliniate la dreapta:

```
cout <<setiosflags (ios::right);
```

Pentru a înţelege mai bine stările indicatorilor I/O, se recomandă examinarea fişierului antet folosit de compilator pentru definiţiile legate de stream-uri (cum sunt IOSTREAM.H, ISTREAM.H, OSTREAM.H). În Capitolul 12 veţi învăţa să creaţi proprii dumneavoastră manipulatori, pentru a simplifica folosirea acestor indicatori în programe.

FACILITĂŢI OFERITE DE FUNCŢIILE MEMBRU ALE STREAM-ULUI DE INTRARE

Programele C++ mai simple folosesc pentru operaţia de intrare fluxul *cin* şi operatorul de inserţie. Pe măsură ce programele devin mai complexe, va fi necesar un control mai precis asupra operaţiilor de intrare, decât cel oferit de operatorul de inserţie. În astfel de situaţii, programele pot folosi funcţiile membru de intrare ale stream-ului I/O, discutate în acest capitol.

DETERMINAREA NUMĂRULUI DE CARACTERE EXTRASE

În momentul extragerii caracterelor dintr-un stream de intrare folosind operatorul de extracţie, uneori este necesară cunoaşterea numărului de caractere extrase. În acest scop, programele pot folosi funcţia membru *gcount*. De exemplu, programul următor, GCOUNT.CPP, vă cere să introduceţi o linie de text.

Apoi, programul citeşte textul folosind funcţia *cin.getline*, iar cu ajutorul funcţiei membru *gcount* determină numărul de caractere citite:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
  char text[64];

  cout << "Type in a line of text and press Enter" << endl;

  cin.getline(text, sizeof(text));

  cout << "You typed: " << text << " " << cin.gcount() <<
       " characters" << endl;
}
```

Observaţie: *La contorizarea caracterelor unui string (şir de caractere), funcţia* gcount *include şi caracterul CR (carriage return - retur de car) - adică, dacă introduceţi şirul 123, valoarea lui* **gcount** *va fi 4.*

Compilaţi şi executaţi acest program. Experimentaţi funcţia *gcount* introducând şiruri cu număr diferit de caractere. Aveţi în vedere că *gcount* include şi caracterul de sfârşit de linie CR.

CITIREA UNEI LINII DE LA TASTATURĂ SAU DE LA STDIN

Când programele realizează operaţii de intrare folosind stream-ul de intrare *cin*, acesta separă valorile prin spaţii albe (spaţii simple, tabulatori sau caractere sfârşit de linie). În funcţie de valorile pe care programele trebuie să le introducă, poate fi necesară citirea unei linii întregi de text, ca apoi părţi ale acesteia să fie prelucrate. Pentru citirea unei linii de text, programele pot folosi funcţia membru *getline*. Următorul program, GETLINE.CPP, foloseşte funcţia membru *getline* pentru citirea unei linii de text de la tastatură.

```
#include <iostream.h>

void main(void)
{
  char line[128];

  cout << "Type in a line of text and press Enter" << endl;
  cin.getline(line, sizeof(line));
  cout << "You typed: " << line << endl;
}
```

În mod prestabilit, funcţia membru *getline* citeşte caracterele până la întâlnirea caracterului sfârşit de linie sau până la un număr specificat de caractere. În funcţie de cerinţele de intrare ale programului, uneori este necesară oprirea operaţiei de intrare la întâlnirea unui anumit caracter. Pentru a termina operaţia de intrare când *getline* întâlneşte litera X, se poate apela funcţia membru astfel:

```
cin.getline (line, sizeof (line), 'X');
```

La stoparea operaţiilor I/O folosind un anumit caracter, este important să ştim că următoarea operaţie de intrare va continua la întâlnirea caracterului imediat următor caracterului de stopare. De exemplu, următorul program STOPON_X.CPP, încheie prima operaţie de intrare pe caracterul X. Programul realizează apoi a doua operaţie de intrare.

```
#include <iostream.h>

void main(void)
{
  char line[128];

  cout << "Type in a line of text and press Enter" << endl;
  cin.getline(line, sizeof(line), 'X');
  cout << "First line: " << line << endl;
  cin.getline(line, sizeof(line));
  cout << "Second line: " << line;
}
```

Compilaţi şi executaţi acest program. Daţi la intrare o serie de cuvinte separate de caracterul X. Programul va afişa textul ce precede caracterul X pe prima linie şi textul ce urmează după X pe a doua linie.

Observaţie: *Caracterul X în acest program este dependent de tipul literei (case-sensitive). Litera mică 'x' nu va determina oprirea intrării la primul apel al funcţiei* **cin.getline**. *De asemenea, până la introducerea caracterului X, puteţi tasta ENTER ori de câte ori doriţi, fără a opri operaţia de intrare la primul apel al lui* cin.getline; *aceasta se va încheia la prima tastare ENTER după introducerea caracterului X.*



FOLOSIREA FUNCŢIILOR MEMBRU ALE STREAM-ULUI CIN

Aşa cum s-a spus, *cin* este un obiect de tipul *istream*. De aceea *cin* suportă mai multe funcţii. Următoarea instrucţiune, de exemplu, foloseşte funcţia membru *getline* pentru a citi o linie de text de la tastatură:

```
cin.getline (line, sizeof (live));
```

Examinând definiţia stream-ului *cin* din fişierul IOSTREAM.H, puteţi determina rapid funcţiile membru disponibile prin vizualizarea prototipurilor acestora.

REALIZAREA OPERAŢIILOR DE INTRARE CARACTER CU CARACTER

În funcţie de cerinţele de intrare ale unui program, uneori sunt necesare operaţii de introducere a câte unui singur caracter. În astfel de situaţii se poate folosi funcţia membru *get*. De exemplu, programul următor, YES_NO.CPP, invită utilizatorul să introducă un răspuns prin una din alternativele Y sau N. Programul foloseşte funcţia membru *get* până când întâlneşte caracterul Y sau N:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
   char letter;

   cout << "Type a Y or N: ";

   do {
     letter = cin.get();              // Read a character
     letter = toupper(letter);        // Convert to uppercase
   } while ((letter != 'Y') && (letter != 'N'));

   cout << endl << "You typed: " << letter << endl;
}
```

Aşa cum se poate vedea, programul ciclează până la întâlnirea unui caracter Y sau N. Pentru simplificarea testului, programul converteşte fiecare literă citită în majusculă.

Reţineţi că operaţiile de intrare ce folosesc stream-ul I/O *cin* nu se pot executa numai de la tastatură. Următorul program, TO_UPPER.CPP, foloseşte funcţia membru *get* pentru a citi câte un caracter de la intrarea redirijată, convertind fiecare literă în majusculă, până când este întâlnit sfârşitul de fişier:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
   char letter;

   while (! cin.eof())
     {
       letter = cin.get();
       letter = toupper(letter);
       cout << letter;
     }
}
```


După cum se vede, programul foloseşte funcţia membru *eof* pentru detectarea sfârşitului de fişier (sau al intrării redirijate). Tot în acest capitol vom examina funcţia *eof* în detaliu.

Următoarea comandă foloseşte comanda TO_UPPER pentru afişarea conţinutului fişierului TO_UPPER.CPP cu litere majuscule:

```
C:\> TO_UPPER < TO_UPPER.CPP <Enter>
```

ANTICIPAREA URMĂTORULUI CARACTER LA INTRARE

În funcţie de cerinţele de intrare ale unui program, uneori este necesară citirea unor caractere până la un anumit caracter, cu excepţia acestuia. De exemplu, să presupunem că un program citeşte un fişier ce conţine nume urmate de numere de telefon, ca mai jos:

Joe Smith 555-1212
John Davis 222-2323
Betty Lou Johnson 333-3343

În acest caz, programul ar putea atribui caracterele ce preced numărul de telefon unei variabile de tip şir. Pentru a citi caracterele până la un anumit caracter, se poate folosi funcţia membru *cin.peek*. Funcţia citeşte un caracter din buffer-ul de intrare, fără a-l scoate din buffer. Următorul program, TO_DIGIT.CPP, vă cere să introduceţi cuvinte, urmate de un număr. Programul foloseşte funcţia membru *peek* pentru a localiza prima cifră, atribuind caracterele care preced numărul unui şir de caractere denumit *line*:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
   char letter, string[128];

   int i = 0, done = 0;

   cout << "Type in a string terminated by a number" << endl;

   do {
     letter = cin.peek();

     if (! isdigit(letter))
       string[i++] = cin.get();
     else
       done = 1;
   } while ((! done) && (i < sizeof(string)));

   string[i] = NULL;

   cout << "String input: " << string << endl;
}
```

***Observaţie:** Majoritatea programelor ce folosesc funcţia membru* **peek** *pot fi rescrise pentru a elimina necesitatea programelor de a folosi funcţia, îmbunătăţindu-le astfel claritatea. Dacă folosiţi în programe funcţia* **peek**, *trebuie să aveţi în vedere şi modalităţi de a rescrie programele.*

READUCEREA UNUI CARACTER ÎN BUFFER-UL DE INTRARE

Programul anterior folosea funcţia membru *peek* a stream-ului de intrare pentru a determina următorul caracter din buffer-ul de intrare. În funcţie de operaţiile efectuate de un program, uneori poate fi necesară plasarea unui caracter în buffer-ul de intrare sau plasarea în buffer a altui caracter decât cel citit. În astfel de situaţii, programele pot folosi funcţia membru *putback*. Următorul program, PUTBACK.CPP, citeşte caractere din buffer-ul de intrare, afişând numărul de caractere citite. Programul contorizează numai caracterele majuscule. La întâlnirea unei litere mici, programul plasează înapoi în buffer majuscula echivalentă:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
 {
   int count[26];
   char i;

   char letter;

   for (i = 0; i < 26; i++)
     count[i] = 0;

   cout << "Type in a string and press Enter" << endl;

   do {
     letter = cin.get();

     if ((letter >= 'a') && (letter <= 'z'))
       cin.putback(toupper(letter));
     else if ((letter >= 'A') && (letter <= 'Z'))
       count[letter - 'A']++;
   } while (letter != '\n');

   for (i = 0; i < 26; i++)
    {
     cout.put((char)('A' + i));
     cout << " " << count[i] << endl;
    }
 }
```

După cum se observă, dacă programul întâlneşte o literă mică, el foloseşte funcţia membru *putback* pentru a plasa litera majusculă echivalentă în buffer-ul de intrare. În acest mod, programul va contoriza numai literele majuscule.

***Observaţie:** Majoritatea programelor care folosesc funcţia membru* putback *pot fi rescrise fără a folosi această funcţie, îmbunătăţindu-se astfel claritatea acestora. Dacă folosiţi în programe funcţia* putback *e bine să vă gândiţi la modalităţi de a rescrie acele programe.*

DETECTAREA SFÂRŞITULUI DE FIŞIER

Capitolul 3 examinează în detaliu operaţiile cu fişiere C++. Veţi afla atunci că multe programe citesc fişierele de la început până la sfârşit, folosind funcţia membru *eof* de detectare a sfârşitului de fişier. Când scrieţi programe ce folosesc redirectarea I/O, puteţi folosi funcţia *eof* pentru depistarea sfârşitului intrării redirectate. De exemplu, următorul program, LINE.CPP, citeşte dintr-o intrare redirectată şi apoi afişează numărul de linii citite:

```
#include <iostream.h>

void main(void)
 {
   char line[256];
   long count = 0;

   while (! cin.eof())
    {
      cin.getline(line, sizeof(line));
      count++;
    }

   cout << "Lines read: " << count << endl;
 }
```

După compilarea acestui program, îi puteţi redirija intrarea, astfel:

```
C:\> LINECNT  <  LINECNT.CPP  <ENTER>
Lines read: 15
```

După cum se vede, acest program, ca şi altele prezentate în acest capitol, ciclează până la întâlnirea sfârşitului de fişier.

IGNORAREA CARACTERELOR DIN STREAM-UL DE INTRARE

La realizarea operaţiilor de I/O, pot exista situaţii când se doreşte ignorarea caracterelor din stream-ul de intrare. De exemplu, să presupunem că un program trebuie să citească următorul fişier şi are nevoie numai de numerele de asigurări sociale:

```
Smith, John     111-22-3333
Lewis, Bill     222-33-4444
Jones, Mary     333-44-5555
```

Folosind funcţia membru *ignore*, se pot ignora caractere din stream-ul de intrare. De exemplu, următoarea instrucţiune dirijează *cin* să ignore următoarele 10 caractere de la intrare:

```
cin.ignore (10);
```

Programele pot, de asemenea, să dirijeze stream-ul I/O pentru a ignora caracterele până la întâlnirea unui anumit caracter. De exemplu, următoarea instrucţiune determină *cin* să ignore primele 10 caractere sau caracterele până la litera A inclusiv:

```
cin.ignore (10, 'A');
```

În acest caz, *cin* va ignora cel mult 10 caractere. Cu alte cuvinte, dacă litera A apare în primele 10 caractere, *cin* nu va mai ignora caracterele ce apar după litera A.

CITIREA ŞI SALVAREA DIFERITELOR VALORI ALE PARAMETRILOR I/O

Aşa cum aţi învăţat, stream-urile I/O furnizează câţiva manipulatori şi funcţii ce permit controlul unor parametri I/O. De exemplu, manipulatorul *setw* permite definirea lăţimii unui câmp de afişare. Când programele folosesc aceşti manipulatori pentru a controla diferite valori ale unor parametri, poate fi necesară determinarea valorilor indicatorilor curenţi, astfel încât programele să poată fi reconstituite mai târziu. În astfel de situaţii, se pot folosi funcţiile din tabela 1.3. pentru citirea valorilor curente.

Funcţia membru	Rezultatul returnat
flags	valoarea indicatorilor curenţi
width	lăţimea câmpului curent de afişare
fill	caracterul de completare
precision	valoarea preciziei curente

Tabela 1.3. Funcţiile membru ale stream-ului I/O ce returnează valori ale parametrilor de stare

Următorul program, SHOWINFO.CPP, foloseşte aceste funcţii pentru a afişa valorile prestabilite ale parametrilor unui stream:

```
#include <iostream.h>

void main(void)
{
   long flags;

   flags = cout.flags();
   cout << "Default flags: " << hex << flags << endl;
   cout << "Default width: " << dec << cout.width() << endl;
```



```
   cout << "Default fill: >" << cout.fill() << "<" << endl;
   cout << "Default precision: " << cout.precision() << endl;
}
```

La compilarea şi execuţia acestui program, pe ecran se va afişa:

```
C:\> SHOWINFO  <ENTER>
Default flags: 2001
Default width: 0
Default fill: > <
Default precision: 6
```



SĂ ÎNŢELEGEM PARAMETRII STREAM-ULUI DE IEŞIRE

C++ dispune de o serie de funcţii membru ce permit controlul spaţierii şi formatării unui stream. Când folosiţi aceste funcţii pentru a schimba o valoare de parametru, schimbarea rămâne activă până la terminarea programului sau efectuarea altei schimbări. Dacă examinaţi fişierul antet folosit de compilator pentru definiţiile de stream-uri, veţi afla că aceste funcţii corespund variabilelor membru ale unei clase de stream I/O. La invocarea funcţiei, valoarea variabilei membru corespunzătoare este schimbată. Ulterior, la realizarea unei operaţii I/O cu acel stream, operaţia va folosi valoarea stabilită a variabilei membru pentru controlul intrării sau ieşirii.

FACILITĂŢI OFERITE DE FUNCŢIILE MEMBRU ALE STREAM-ULUI DE IEŞIRE

Aşa cum aţi învăţat, stream-urile de intrare furnizează câteva funcţii membru pentru realizarea unor operaţii I/O specifice. Acest paragraf examinează funcţii membru ale stream-ului de ieşire cu aceeaşi finalitate. De exemplu, următorul program, PUTALPHA.CPP, foloseşte funcţia membru *put* pentru a afişa pe ecran literele alfabetului, una câte una:

```
#include <iostream.h>

void main(void)
{
   char letter;

   for (letter = 'A'; letter <= 'Z'; letter++)
     cout.put(letter);
}
```

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> PUTALPHA  <ENTER>
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Poate vă întrebaţi de ce în programul precedent s-a folosit funcţia membru *put*, în loc de *cout*, ca mai jos:

```
for (letter = 'A'; letter <= 'Z'; letter++)
  cout << letter;
```

Dacă în program variabila *letter* este declarată de tip *char*, se pot folosi ambele tehnici pentru afişarea literelor. Însă dacă variabila *letter* este de tip *int*, atunci prin a doua metodă pe ecran se vor afişa numerele între 65 şi 90, în loc de literele A până la Z.

CHEIA SUCCESULUI

FOLOSIREA FUNCŢIILOR MEMBRU ALE STREAM-ULUI COUT

Aşa cum s-a văzut, *cout* este un obiect de tip *ostream*. Ca atare, *cout* suportă mai multe funcţii membru. Următoarea instrucţiune, de exemplu, foloseşte funcţia membru *put* pentru a scrie un caracter la dispozitivul standard de ieşire:

```
cout.put(7);// semnal sonor prin difuzorul incorporat
```

Prin examinarea definiţiei clasei *cout* din fişierul IOSTREAM.H, se pot găsi rapid funcţiile membru disponibile, prin consultarea listei de prototipuri.

TESTAREA REUŞITEI OPERAŢIILOR I/O

Fiecare din programele anterioare au realizat operaţii de I/O, presupunând că acele operaţii s-au derulat cu succes. Când programele scriu mesaje simple pe ecran sau citesc una-două linii de la tastatură, în mod normal putem presupune că operaţiile se încheie cu succes. Însă, pe măsură ce programele devin mai complexe, ar trebui verificată finalitatea operaţiilor, folosind funcţiile membru *good, bad, fail* şi *rdstate*. Fiecare din aceste funcţii membru examinează biţii de stare din indicatorul de stare al stream-ului I/O, pentru a determina dacă a avut loc o eroare. În funcţie de compilatorul folosit, valoarea şi semnificaţia biţilor de stare poate fi uşor diferită. Totuşi, dacă examinaţi fişierul antet al definiţiilor legate de stream-uri, veţi găsi o definiţie de tip enumerativ a acestor biţi, ca mai jos:

```
enum io_state {
  goodbit = 0x00   // Set if I/O operations have been successful
  eofbit = 0x01,   // Set if currently at the end of the file
  failbit = 0x02,  // Set if last I/O operation failed
  badbit = 0x04,   // Set if an invalid operation was attempted
  hardfail = 0x80  // Set if an unrecoverable error occurred
};
```

Pentru a vedea dacă a avut loc o eroare I/O, programele pot testa diferiţi biţi de stare. De exemplu, următorul program, TEST_IO.CPP, foloseşte funcţia membru *fail* pentru a determina dacă operaţiile I/O s-au încheiat cu succes:


```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>

void main(void)
{
  char letter;

  while (! cin.eof())
   {
     letter = cin.get();
     if (cin.fail())
      {
        cerr << "Error reading input" << endl;
        exit(1);
      }

     cout.put((char)toupper(letter));
     if (cout.fail())
      {
        cerr << "Error writing output" << endl;
        exit(1);
      }
   }
}
```

Să presupunem, de exemplu, că programul precedent va fi apelat astfel:

```
C:\> TEST_IO < FILENAME.CPP > A:UPPER.CPP <ENTER>
```

În acest caz, TEST_IO vă citi conţinutul fişierului FILENAME.CPP, scriind majuscule echivalente fiecărui caracter pe fişierul UPPER.CPP de pe unitatea A. Dacă apare o eroare când TEST_IO citeşte fişierul de intrare sau scrie fişierul de ieşire (de pildă, spaţiu insuficient pe disc), programul va afişa un mesaj de eroare în stream-ul fişierului standard de erori şi se va termina. Deoarece programele pot testa corectitudinea operaţiilor I/O în mod obişnuit, stream-urile I/O definesc semnul de exclamare ca un operator ce detectează o operaţie I/O nereuşită. Astfel, următoarele instrucţiuni *if* sunt identice din punct de vedere funcţional:

```
if (cin.fail())                  if (! cin)
```

În cazul când un stream I/O întâlneşte o eroare, bitul de eroare rămâne setat până când se şterge indicatorul prin funcţia membru *clear*, cum se arată mai jos:

```
if (cout.fail())
{
  cerr << "Display some error message" << endl;
  cout.clear();
}
```

În funcţie de modul în care un program manipulează erorile I/O, se poate cere examinarea indicatorilor de stare în vederea determinării tipului de eroare apărută. Pentru aceasta se poate folosi funcţia membru *rdstate*, ca mai jos:

```
io_state = cin.rdstate ();
```

Pentru a înţelege mai bine definiţiile acestor funcţii membru, puteţi examina fişierul antet folosit de compilator pentru definiţiile legate de stream-urile I/O.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM STAREA UNUI STREAM I/O

La realizarea operaţiilor cu stream-urile I/O, se poate testa în programe finalizarea cu succes a unei operaţii folosind funcţiile membru ale clasei I/O. Dacă examinaţi fişierul antet pentru definiţiile legate de stream-uri, veţi afla că stream-urile I/O conţin o variabilă membru de stare ai cărei biţi corespund stării curente a stream-ului. Analizând biţii acestei variabile, programele pot determina încheierea cu succes a operaţiei I/O anterioare. Majoritatea programelor, totuşi, folosesc funcţii membru incorporate pentru a realiza aceste teste.

INTRĂRI-IEŞIRI CU ŞI FĂRĂ BUFFER

Stream-urile I/O *cout* şi *clog* folosesc ieşirea prin buffer, adică ieşirea nu este afişată decât când buffer-ul se umple, programul se încheie, buffer-ul este golit intenţionat , sau, în cazul stream-ului *cout*, când programul citeşte din stream-ul *cin*. Următorul program, BUFFERED.CPP, scrie ieşirea la *cout* şi aşteaptă apoi 3 secunde până la terminare. Când programul ia sfârşit, buffer-ul de ieşire este afişat, ca în următorul exemplu:

```
#include <iostream.h>
#include <time.h>

void main(void)
 {
   time_t start_time, current_time;

   cout << "Hello C++ world!" << endl;
   time(&start_time);

   do {
     time(&current_time);
   } while ((current_time - start_time) < 3);
 }
```

Programul foloseşte funcţia *time* pentru determinarea timpului curent şi introduce o întârziere de 3 secunde.

GOLIREA UNUI BUFFER DE IEŞIRE

În funcţie de operaţiile efectuate de program apar situaţii când buffer-ul de ieşire trebuie golit înainte de a se umple, înainte de încheierea programului, sau înainte de a realiza o operaţie de intrare. În astfel de situaţii programele pot folosi funcţia membru *flush*. De exemplu, următorul program, FLUSH.CPP, scrie un mesaj la *cout*, face o pauză de 2 secunde, goleşte stream-ul de ieşire şi apoi mai întârzie 3 secunde până la încheiere:

```
#include <iostream.h>
#include <time.h>

void main(void)
 {
   time_t start_time, current_time;

   cout << "Hello C++ world!" << endl;
   time(&start_time);

   do {
     time(&current_time);
   } while ((current_time - start_time) < 2);

   cout.flush();

   time(&start_time);

   do {
     time(&current_time);
   } while ((current_time - start_time) < 3);

   cout << "Done!" << endl;
 }
```

În Capitolul 11 veţi aprofunda stream-urile I/O prin buffer când veţi examina clasele *streambuf* şi *filebuf*.

FOLOSIREA FUNCŢIILOR MEMBRU PENTRU CONTROLUL INDICATORILOR DE STREAM I/O

Aţi învăţat ceva mai devreme, în acest capitol, cum se controlează ieşirea unui stream I/O folosind manipulatorii *setiosflags* şi *resetiosflags*. Pe lângă aceşti manipulatori, programele pot avea un control mai precis asupra parametrilor acestor indicatori folosind funcţiile membru ale stream-urilor I/O ce vor fi discutate în continuare. De exemplu, următorul program, IOFLAGS.CPP, foloseşte funcţia membru *flags* pentru afişarea valorilor parametrului indicatorului de stream:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
 {
   cout << "cout starting flag settings: " <<
          hex << cout.flags() << endl;
   cout << setiosflags(ios::showbase);
   cout << "cout ending flag settings: " << cout.flags() << endl;
 }
```

După compilarea şi execuţia acestui program, pe ecran va apărea un rezultat dependent de compilatorul folosit, având următoarea formă generală:

```
C:\> IOFLAGS  <ENTER>
cout starting flag settings: 1
cout ending flag settings: 0xc1
```

Folosind funcţia membru *flags*, un program poate salva valorile unui indicator la diferite momente, pentru a putea fi ulterior readuse la forma iniţială. De exemplu, programul SAVEFLAG.CPP, foloseşte funcţia membru *flags* pentru a citi valoarea curentă a unui indicator şi a o restabili ulterior.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   long flag_settings;

   flag_settings = cout.flags();  // Read the flags

   cout << "Hexadecimal values: " << hex << 255 << " " <<
          10 << endl;

   cout.flags(flag_settings);  // Restore the flags

   cout << "Decimal values: " << 255 << " " << 10 << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SAVEFLAG  <ENTER>
Hexadecimal values: ff a
Decimal values: 255 10
```

Pe lângă folosirea funcţiei membru *flags* pentru configurarea indicatorilor stream-urilor I/O, programele mai pot folosi şi funcţiile membru *setf* şi *unsetf*. Pentru a crea şi a şterge parametrii indicatorilor folosind aceste două funcţii membru, trebuie cunoscuţi biţii ce corespund fiecărui indicator. Prin folosirea tipului de enumerare al indicatorilor prezentaţi anterior în acest capitol, se pot determina valorile corespunzătoare ale biţilor. De exemplu, următorul program, SETCLEAR.CPP, foloseşte funcţiile membru *setf* şi *unsetf* pentru a crea şi a şterge indicatorii stream-ului I/O:

```
#include <iostream.h>
#include <iomanip.h>
```



```
void main(void)
 {
   cout.setf(0x80);      // Show base
   cout << hex << 255 << " " << 10 << endl;

   cout.unsetf(0x80);    // Clear the base display
   cout << hex << 255 << " " << 10 << endl;
 }
```

Programul foloseşte funcţia membru *setf* pentru a activa afişarea în bază hexazecimală. După ce afişează două valori, programul foloseşte funcţia membru *unsetf* pentru a dezactiva afişarea în baza respectivă. La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> SETCLEAR  <ENTER>
0xff 0xa
ff a
```

REZUMAT

Fiecare program pe care îl creaţi, indiferent de destinaţia sa, va folosi operaţii de intrare sau ieşire. De aceea, înţelegerea stream-urilor I/O şi a posibilităţilor lor sunt esenţiale pentru succesul în programarea C++. Înainte de a continua, în Capitolul 2, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Imaginea optimă a unui stream I/O este o serie de octeţi.
- ✓ Compilatorul foloseşte un fişier antet pentru a defini stream-urile I/O – ios, istream şi ostream, precum şi funcţiile şi variabilele membru. Tipăriţi o copie a acestui fişier şi studiaţi-o cu atenţie.
- ✓ Operatorii de inserţie (<<) şi de extracţie (>>) permit programelor să introducă şi să extragă caractere în/din stream-ul de I/O.
- ✓ Manipulatorii sunt elemente pe care le puteţi plasa într-un stream de intrare sau de ieşire pentru a controla formatarea I/O. Fişierul antet IOMANIP.H defineşte manipulatorii disponibili.
- ✓ Când un program foloseşte un manipulator pentru a configura formatarea unui stream, C++ stabileşte valoarea biţilor din câmpul indicatorului de stream ce controlează configurarea. Pentru a înţelege mai bine acea configuraţie de biţi, consultaţi fişierul antet folosit de compilator la definirea stream-urilor.
- ✓ Stream-urile de intrare/ieşire conţin diferite metode (funcţii) ce permit controlul formatului ieşirii sau realizarea unei anumite operaţii I/O. Compilatorul defineşte prototipurile funcţiilor membru ale stream-ului într-un fişier antet.
- ✓ Stream-urile de ieşire *cout* şi *clog* realizează ieşirea prin buffer, ceea ce înseamnă că ieşirea este scrisă numai atunci când buffer-ul se umple, programul se încheie, programul goleşte buffer-ul, sau, în cazul lui *cout*, când programul realizează o operaţie de intrare.

CAPITOLUL 2
ACOMODAREA CU CLASE ŞI OBIECTE

Este posibil să puteţi scrie o mulţime de programe C++ fără a înţelege cu adevărat sau fără a folosi obiecte, deşi obiectele şi clasele sunt esenţiale pentru utilizarea la maximum a limbajului. Dar nu vă speriaţi! Acest capitol începe cu primii paşi şi prezintă regulile de bază ce trebuie cunoscute. Capitolele următoare adaugă concepte pe baza informaţiilor prezentate aici. În acest mod, puteţi învăţa pe rând părţile importante ale noţiunii de obiect. Ca şi mai înainte, trebuie să aveţi răbdarea de a experimenta programele prezentate aici. Chiar şi prin schimbări simple, puteţi învăţa foarte mult. La încheierea terminării capitolului, veţi înţelege următoarele:

- Ce este un obiect
- Cum un program complicat, de mari dimensiuni, este mai uşor de conceput şi de redactat folosind o abordare orientată pe obiect în locul uneia convenţionale, orientată pe funcţie
- Cum un obiect poate economisi timp şi efort
- Ce este moştenirea
- Avantajul folosirii obiectelor
- Cum se creează un obiect
- Avantajele şi dezavantajele funcţiilor *inline*
- Cum se folosesc două variabile diferite cu acelaşi nume în acelaşi program
- Ce sunt funcţiile constructor şi destructor de clase
- Cum se atribuie valoarea unui obiect altui obiect

PRIMUL CONTACT CU PROGRAMAREA ORIENTATĂ PE OBIECTE

În cel mai simplu sens, un obiect este un lucru. Un câine, o carte, chiar un computer, toate sunt obiecte. În trecut, programatorii îşi concepeau programele ca pe liste lungi de instrucţiuni pentru realizarea unei sarcini specifice. La crearea

de programe orientate pe obiecte, accentul se pune pe obiectele componente ale programului.

Să presupunem, de exemplu, că scrieţi un program ce implementează simplu un procesor. Dacă vă gândiţi la toate funcţiile realizate de un procesor de texte, veţi fi repede copleşit de acestea. În schimb, dacă vă imaginaţi procesorul de cuvinte ca o colecţie de obiecte distincte, programul va deveni mai puţin intimidant. În figura 2.1 sunt ilustrate obiectele principale ale sistemului de procesare de texte.

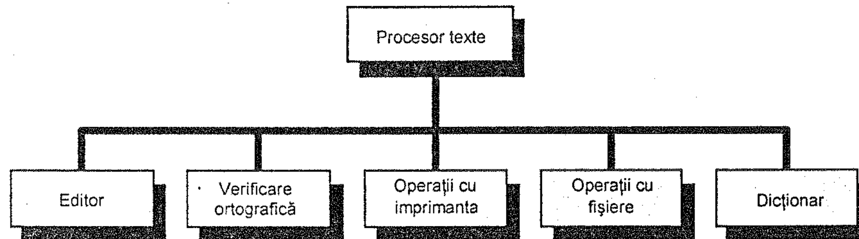


Figura 2.1. - Structura unui procesor de cuvinte ca o colecţie de obiecte.

Pe măsură ce examinaţi fiecare obiect, veţi descoperi că acesta este, de asemenea, compus din obiecte, cum se arată în Figura 2.2.

Când începeţi să identificaţi obiectele din sistemul dvs., veţi vedea că diferite părţi ale programului folosesc acelaşi tip de obiecte. De aceea, scriind programul vostru în termeni de obiecte, veţi putea uşor (şi repede) să *refolosiţi* codul scris pentru o anumită secvenţă într-o altă parte a programului, sau chiar în alt program. În aceasta rezidă o parte din puterea limbajului C++.

Primul pas în crearea programelor orientate pe obiecte este de a identifica obiectele ce formează sistemul. Pentru a înţelege mai bine cum se identifică obiectele sistemului, puteţi face referinţă la următoarele cărţi de analiză şi proiectare orientate pe obiecte:

- *Object-Oriented Design and Applications* - Booch, Benjamin/Cummings, 1991
- *Object-Oriented Analysis*, Coad & Yourton, Yourdon Press, 1990
- *Object Data Management*, Cattel, Addison-Wesley, 1991
- *Object-Oriented Reuse, Concurrency, and Distribution*, Atkinson, Addison-Wesley, 1991
- *Object-Oriented Methods*, Graham, Addison-Wesley, 1991
- *An Introduction to Object-Oriented Programming*, Budd, Addison-Wesley, 1991

- *Object-Oriented Software Construction*, Meyer, Prentice-Hall, 1988
- *Object-Oriented System Analysis: A Model-Driven Approach*, Embley, Kurtz and Woodfield, Yourdon Press, 1992

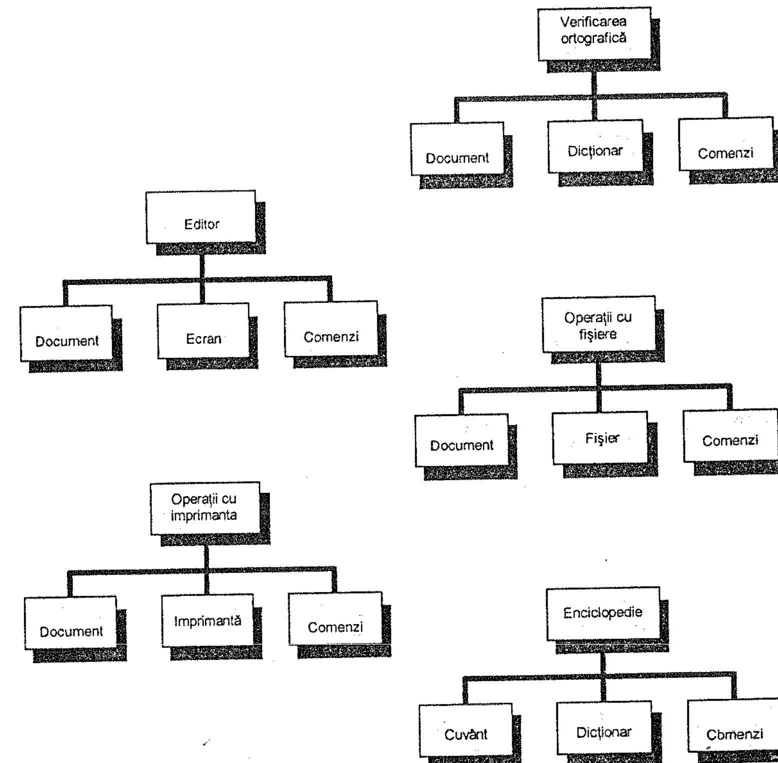


Figura 2.2. Identificarea obiectelor adiţionale în cadrul procesorului de cuvinte.

După identificarea obiectelor, trebuie să determinaţi scopul fiecărui obiect. Pentru aceasta, gândiţi-vă la operaţiile realizate de un obiect sau la operaţiile efectuate pe acel obiect. De exemplu, dacă *fişier* este un obiect, un program poate copia, şterge sau redenumi fişierul. Este important de observat că, în general, aceste operaţii se aplică fiecărui fişier de pe disc, indiferent de conţinutul său. Aceste operaţii vor deveni *funcţiile membru* ale obiectului, pentru care ulterior veţi scrie funcţii C++ în programul respectiv. Apoi trebuie identificate informaţiile despre obiect. În cazul unui obiect fişier, trebuie cunoscute numele fişierului, dimensiunea, gradul de protecţie, eventual data la care fişierul a fost creat sau modificat ultima oară. Aceste date vor deveni *variabile membru* ale obiectului. Conceptual, obiectul fişier se poate vedea ca în Figura 2.3.

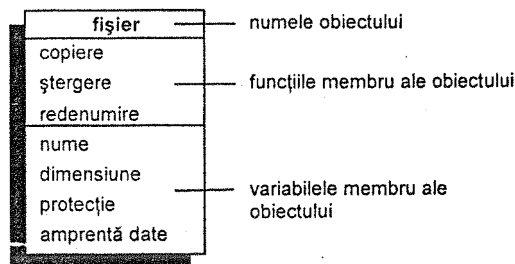


Figura 2.3. Funcţiile şi variabilele membru ale unui obiect fişier.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM PROGRAMAREA ORIENTATĂ PE OBIECTE

Programarea orientată pe obiecte este bazată pe scrierea de programe cu referire la obiectele ce intră în componenţa unui sistem. În cadrul unui sistem, obiectele memorează tipuri specifice de informaţii şi execută operaţii specifice pe acele informaţii. Primul pas în crearea unui program orientat pe obiecte este identificarea obiectelor sistemului, informaţiile de bază referitoare la fiecare obiect şi operaţiile realizate pe obiecte. La scrierea primelor programe în C++, construcţia lor se va opri, probabil, aici. Pe măsură ce deveniţi mai experimentat, veţi căuta relaţii între obiecte ce vor permite construcţia unor obiecte noi din altele mai vechi. Pentru moment, însă, dacă cineva vă cere să definiţi un obiect C++, spuneţi-le pur şi simplu că un obiect reprezintă o entitate a lumii reale, poate conţine date (variabile membru) şi are un set concret de operaţii (funcţii membru) ce se efectuează asupra acestora.

Uneori este necesară afişarea sau tipărirea unor fişiere; alteori, fişierele conţin programe executabile de care aveţi nevoie. Ca atare, la definiţia obiectelor trebuie adăugate metodele de efectuare a operaţiilor respective, ceea ce nu este greşit.Totuşi, există o variantă mai bună de a prelucra cazurile speciale.

De exemplu, dacă fişierul conţine un document, conţinutul acestui fişier poate fi tipărit, operaţie care nu ar avea sens dacă fişierul ar conţine un program executabil; în această ultimă situaţie execuţia conţinutului fişierului este operaţia dorită. La încercarea de a executa conţinutul unui fişier document va rezulta o eroare. Soluţia constă în identificarea a două noi clase obiect, denumite *document* şi *program*. O modalitate de a crea aceste noi clase este adăugarea componentelor respective la fiecare clasă, aşa cum se arată în Figura 2.4.

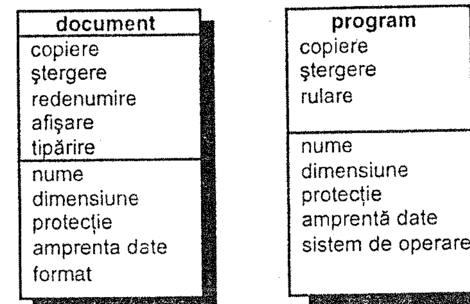


Figura 2.4. Crearea unor clase document şi program independente.

După cum se poate vedea, la obiectul *document* se adaugă funcţiile membru *display* şi *print* şi variabila membru *format* (care specifică formatul intern al documentului, cum ar fi Word sau WordPerfect). În acelaşi mod, la obiectul *program* se adaugă funcţia membru *run* şi câmpul *sistem de operare*, care specifică dacă programul rulează sub DOS, WINDOWS sau UNIX.

Aceste două noi clase practic dublează clasa *fişier* definită anterior. De aceea o soluţie mai bună este de a construi cele două clase bazate pe clasa *fişier*, cum se arată în Figura 2.5.

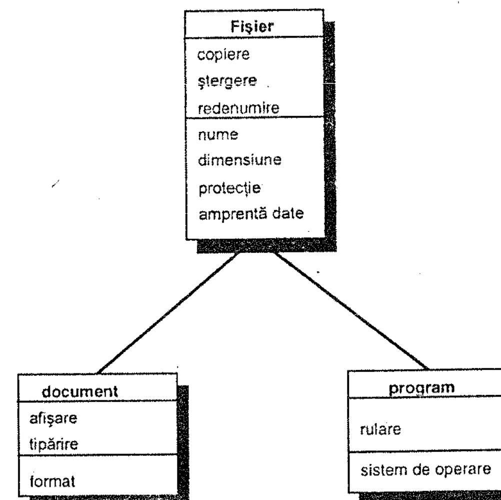


Figura 2.5. Crearea noilor clase pe baza clasei fişier.

Când programul dvs. construieşte o clasă derivată dintr-o altă clasă, noua clasă *moşteneşte* funcţiile şi variabilele membru ale clasei de bază. În cazul claselor *document* şi *program*, obiectele create cu aceste clase pot să folosească nu

numai propriile funcţii şi variabile membru, ci şi pe acelea ale clasei *fişier*. În acest mod, programul dvs. poate să execute cu uşurinţă afişarea, tipărirea, copierea, redenumirea sau ştergerea unui fişier document.

CHEIA SUCCESULUI

SĂ SIMPLIFICĂM DEFINIŢIILE UNEI CLASE

Pe măsură ce definiţi clasele, încercaţi să daţi definiţiilor un caracter general. Cu alte cuvinte, variabilele şi funcţiile membru care apar în clasa de bază trebuie să fie folosite de fiecare din clasele derivate (construite) din aceasta. Dacă este nevoie să adăugaţi la o clasă mai multe variabile şi funcţii membru, puteţi defini o altă clasă, mai specializată, bazată pe clasa generală.

Construcţia claselor în acest mod este esenţa *conceptului de moştenire a claselor*, concept ce va fi tratat în detaliu în Capitolul 4. S-ar putea să vă miraţi de ce e nevoie de acest procedeu. Primul motiv pentru folosirea moştenirii este *refolosirea*, adică posibilitatea ca un program să folosească codul scris şi testat pentru alt program. De exemplu, să presupunem că aţi creat clasa *fişier* pentru un program de buget. Întrucât clasa există, aceasta poate fi rapid folosită în interiorul unor programe noi. Existenţa unei clase nu numai că diminuează efortul de programare, dar reduce şi volumul testărilor necesare, întrucât funcţiile membru ale clasei *fişier* au fost testate şi verificate anterior pentru programul de buget.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM MOŞTENIREA

Moştenirea este capacitatea unei clase *derivate* de a *moşteni* funcţiile şi variabilele membru ale unei clase *de bază* existente. Pe măsură ce identificaţi şi examinaţi obiectele ce compun un sistem, veţi găsi deseori relaţii şi asemănări între obiecte. În multe cazuri, aceste relaţii vă permit să construiţi un obiect din altul, care conţine toate metodele originalului. Când proiectaţi clasele obiect, încercaţi să daţi claselor un caracter cât mai general. Dacă numărul funcţiilor şi variabilelor incluse în definiţia unei clase creşte, atunci posibilitatea de a refolosi acea clasă, sau de a deriva alte clase din aceasta, scade.

INGINERIA PROGRAMELOR ŞI FOLOSIREA OBIECTELOR

De ce ar trebui să ne bazăm pe obiecte în activitatea de programare? Există câţiva termeni de ingineria programelor care sunt folosiţi frecvent pentru a descrie acest lucru. Cu toate că există dezacorduri în domeniul ingineriei de programe în privinţa folosirii optime a obiectelor, sunt clar recunoscute câteva avantaje oferite de acestea:

Uşurinţa proiectării şi refolosirii codului. O dată ce codul funcţionează corect, folosirea obiectelor măreşte posibilitatea de a refolosi codul creat într-o aplicaţie pentru o altă aplicaţie.

Fiabilitate crescută. Deoarece bibliotecile de obiecte au fost anterior testate, folosirea codurilor existente (de lucru) va îmbunătăţi fiabilitatea programului.

Uşurinţa înţelegerii. Permiţând proiectanţilor şi programatorilor să se concentreze asupra părţilor componente ale unui sistem şi furnizând un cadru în care proiectanţii pot identifica obiectele, operaţiile efectuate cu acestea precum şi informaţiile pe care un obiect trebuie să le conţină, utilizarea obiectelor ajută programatorii să înţeleagă componentele importante ale unui sistem.

Grad înalt de abstractizare. Abstractizarea permite programatorilor să aibă o vedere de ansamblu, adică să ignore ansamblul temporar detaliile nesemnificative şi să lucreze cu elementele de sistem ce sunt mai uşor de înţeles. De exemplu, prin concentrarea asupra obiectelor procesorului de cuvinte din exemplul anterior, implementarea acestuia a devenit mai puţin stresantă.

Grad ridicat de încapsulare. *Încapsularea* grupează toate părţile unui obiect într-un modul compact. De exemplu, clasa *fişier* definită în exemplele anterioare combină funcţii şi câmpuri de date cu care programul lucrează într-un fişier. Programatorul care lucrează cu clasa *fişier* nu trebuie să cunoască fiecare componentă a clasei, ci numai cum se foloseşte clasa în program. Clasa, la rândul ei, include toate componentele necesare.

Ascunderea informaţiei. *Ascunderea informaţiei* reprezintă capacitatea unui program de a trata o funcţie, procedură, sau chiar un obiect, ca pe o „cutie neagră“, adică de a le folosi în anumite operaţii fără a cunoaşte structura lor interioară. În Capitolul 1, de exemplu, programele au folosit obiecte tip stream I/O pentru intrare şi ieşire, fără a necesita înţelegerea modului de funcţionare a stream-urilor I/O.

Pe măsură ce vom examina conceptele de programare C++ în această carte, veţi învăţa care sunt legăturile dintre aceste concepte şi definiţiile de mai sus.

SĂ ÎNŢELEGEM OBIECTELE ŞI CLASELE

În decursul discuţiei precedente, termenii *obiecte* şi *clase* au fost folosiţi foarte vag. În general, o *clasă* furnizează un şablon folosit de program la crearea obiectelor. De exemplu, clasa *fişier* descrisă anterior specifică variabilele şi funcţiile membru ce vor fi folosite pentru obiectele *fişier* pe care programul le va crea ulterior. *Un obiect*, aşadar, este o *instanţă* (instance) a unui şablon de

clasă. Cu alte cuvinte, un obiect este o entitate, un exemplu specific, un articol cu care programul lucrează.

Definiţia unei clase este similară unei definiţii de structură în C. De exemplu, următoarele instrucţiuni definesc clasa *fişier*:

```
class file {
public:
  char filename[64];
  long size;
  unsigned protection;
  long datetime;
  int copy(char *target_name);
  int rename(char *target_name);
  int delete_file(void);
};
```

După cum se poate vedea, definiţia clasei indică funcţiile şi variabilele membru. Observaţi eticheta *public* ce apare la începutul clasei. Singurul mod în care programele pot avea acces direct la membrul unei clase, fie aceasta funcţie sau variabilă, este ca acel membru să fie precedat de eticheta *public*. Mai târziu, în acest capitol, veţi învăţa cum să folosiţi etichetele *public* şi *private* pentru a controla modul în care programele pot avea acces la membrii unei clase. În Capitolul 4, veţi învăţa cum clasele condiţionate folosesc eticheta *protected* pentru a furniza o cale intermediară de acces la membri. Deocamdată, însă, plasaţi eticheta *public* la începutul fiecărei clase; astfel toţi membrii clasei vor deveni disponibili în tot programul.

O clasă defineşte un şablon pentru crearea ulterioară a obiectelor. Clasa însăşi nu creează un obiect. Pentru a crea un obiect, se declară variabile de tip obiect, cum este arătat mai jos:

```
void main(void)
 {
    file source, target;    // Declare two file objects

    // Other statements here
 }
```



DIFERENŢA ÎNTRE CLASE ŞI OBIECTE

Dacă citiţi articole şi cărţi despre C++ şi programarea orientată pe obiecte, veţi întâlni termenii *clasă* şi *obiect*. O *clasă* furnizează un şablon ce defineşte funcţiile şi variabilele membru necesare obiectelor având tipul clasei. Un *obiect*, pe de altă parte, este o *instanţă*, un exemplu concret, de fapt o variabilă obiect. O clasă trebuie definită înainte de declararea obiectului.

Pentru a declara o variabilă obiect, specificaţi tipul clasei, urmat de numele variabilei obiect, ca mai jos:

```
nume_clasa      nume_obiect;
```

Procesul creării unui obiect este adesea denumit *instanţierea unui obiect* sau *crearea unei mostre de obiect*.

ANALIZA UNUI EXEMPLU COMPLET

Cel mai bun mijloc de a înţelege clasele şi obiectele C++ este de a crea un program simplu. Următorul program, MOVIES.CPP, creează o clasă denumită *movie* apoi creează 2 obiecte de tip movie, cu numele *fugitive* şi *sleepless*. Programul defineşte clasa după cum urmează:

```
class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
};
```

(Variabile membru → name, first_star, second_star; Funcţii membru → show_movie, initialize)

După cum se poate vedea, clasa *movie* foloseşte trei variabile membru şi două funcţii membru. După definiţia clasei, programul trebuie să definească funcţiile membru *show_movie* şi *initialize*, astfel:

```
void movie::show_movie(void)
 {
  cout << "Movie name: " << name << endl;
  cout << "Starring: " << first_star << " and " << second_star <<
         endl << endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
  strcpy(name, movie_name);
  strcpy(first_star, first);
  strcpy(second_star, second);
 }
```

Definiţia funcţiilor clasei este asemănătoare definiţiei funcţiilor standard, existând totuşi două diferenţe. Mai întâi, numele funcţiilor sunt precedate de numele clasei şi de simbolul „::"

(Nume clasă → movie; Nume funcţie → initialize)

```
void movie::initialize(char *movie_name, char *first, char *second)
```

În al doilea rând, instrucţiunile din corpul unei funcţii ale unei clase pot face referinţă directă la variabilele membru ale clasei:

```
void movie::initialize(char *movie_name, char *first, char *second)
{
  strcpy(name, movie_name);
  strcpy(first_star, first);
  strcpy(second_star, second);
}
```

Nume variabilă locală funcţie

Nume membri clasă

Următoarele instrucţiuni implementează programul MOVIES.CPP:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
          endl << endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
   strcpy(name, movie_name);
   strcpy(first_star, first);
   strcpy(second_star, second);
 }

void main(void)
 {
   movie fugitive, sleepless;

   fugitive.initialize("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
     "Meg Ryan");

   fugitive.show_movie();
   sleepless.show_movie();
 }
```


După cum se poate vedea, programul creează două obiecte de tipul *movie*:

Nume clasă

```
movie fugitive, sleepless;
```

Nume obiecte

Accesul la membrii unei clase este asemănător accesului la componentele unei structuri în C: se specifică numele obiectului, urmat de operatorul '.' şi de numele membrului corespunzător. De exemplu, pentru a apela funcţia *initialize*, obiectul *fugitive* foloseşte operatorul '.' şi numele funcţiei membru, cum este arătat mai jos:

Nume obiect

Parametri funcţie

```
fugitive.initialize("The Fugitive", "Harrison Ford", "Tommy Lee Jones");
```

Nume funcţie

În acest caz, programul a folosit funcţia membru *initialize* pentru a iniţializa variabilele membru ale clasei. Mai târziu în acest capitol veţi învăţa cum să folosiţi funcţiile *constructor* pentru a iniţializa variabilele membru într-un mod mai natural.



CÂND SE FOLOSESC CLASELE ŞI CÂND STRUCTURILE

Dacă aţi programat în C, probabil aţi recunoscut faptul că sintaxa pentru lucrul cu clase este similară celei folosite în C pentru structuri. V-aţi putea întreba, când trebuie să folosiţi clase, şi nu structuri sau uniuni. După cum ştiţi, acestea permit programului să memoreze date dependente. Programele dvs. ar trebui să folosească clase ori de câte ori asupra datelor au loc operaţii concrete. De exemplu, dacă aveţi nevoie de o dată, atunci puteţi folosi o structură. Dar dacă doriţi ca programul să formateze şi să afişeze data, să treacă data într-un fişier, sau să compare două date, este necesară utilizarea unei clase. Similar, dacă trebuie să optaţi între o structură şi o uniune, decizia trebuie luată în funcţie de numărul de valori pe care structura de date le conţine la un moment dat. În sfârşit, ţineţi cont că, în mod prestabilit, membrii unei clase sunt de tip *private*, iar membrii unei uniuni sau structuri sunt de tip *public*.

Dacă experimentaţi structurile din C++, veţi descoperi că ele suportă multe din conceptele clasei din C++, precum date publice şi private, funcţii membru etc. Ca o regulă, dacă creaţi obiecte, atunci folosiţi clase.

ACCESUL LA MEMBRII UNEI CLASE

În programul precedent, aţi folosit operatorul de acces '.' pentru a apela funcţiile membru *initialize* şi *show_movie*. Când programul plasează membrii unui obiect după eticheta *public*, el poate avea acces la membrii acestuia

folosind operatorul '.', De exemplu, următorul program, PUBLIC.CPP, foloseşte funcţia de iniţializare pentru a atribui valori membrilor obiectelor *fugitive* şi *sleepless*. Programul afişează apoi valorile diferiţilor membri, făcând referinţă la membri prin operatorul '.':

```
#include <iostream.h>
#include <string.h>

class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star << endl <<
endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
   strcpy(name, movie_name);
   strcpy(first_star, first);
   strcpy(second_star, second);
 }

void main(void)
 {
   movie fugitive, sleepless;

   fugitive.initialize("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
     "Meg Ryan");

   cout << "The last two movies I've watched are: " <<
           fugitive.name << " and " << sleepless.name << endl;

   cout << "I thought " << fugitive.first_star << " was great!" <<
           endl;
 }
```

Deoarece membrii clasei sunt *public*, programul are acces direct la aceştia. Când compilaţi şi executaţi acest program, ecranul va afişa următoarele:


```
C:\> PUBLIC <ENTER>
The last two movies I've watched are: The Fugitive and Sleepless in Seattle
I thought Harrison Ford was great!
```

Când o clasă defineşte variabilele membru ca fiind *public*, programul le poate referi folosind operatorul '.'. Totuşi, aşa cum veţi afla mai târziu, acest acces direct la variabilele membru nu este întotdeauna de dorit.

FOLOSIREA FUNCŢIILOR INLINE

Aşa cum aţi învăţat, o clasă conţine variabile şi funcţii membru. Când definiţi funcţiile unei clase, aveţi două opţiuni. Mai întâi, puteţi defini funcţiile în afara definiţiei clasei, ca mai jos:

```
class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
          endl << endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
   strcpy(name, movie_name);
   strcpy(first_star, first);
   strcpy(second_star, second);
 }
```

În acest caz, definiţia clasei trebuie să conţină prototipuri ce descriu fiecare funcţie membru a clasei:

```
class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
};
```

Prototipuri de funcţii într-o clasă

De asemenea, definiţiile de funcţii trebuie să specifice numele clasei înaintea numelui funcţiei:


```
void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
           endl << endl;
 }
```

În al doilea rând, se pot defini funcţiile membru ale unei clase în interiorul clasei, scriind instrucţiunile corpului funcţiei în cadrul declaraţiei clasei. De exemplu, programul următor, INLINE.CPP, defineşte funcţiile membru ale clasei în interiorul declaraţiei de clasă:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void)
   {
    cout << "Movie name: " << name << endl;

    cout << "Starring: " << first_star << " and " <<
           second_star << endl << endl;
   }

  void initialize(char *movie_name, char *first, char *second)
   {
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
   }
};

void main(void)
 {
  movie fugitive, sleepless;

  fugitive.initialize("The Fugitive", "Harrison Ford",
    "Tommy Lee Jones");

  sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
    "Meg Ryan");
```



```
  cout << "The last two movies I've watched are: " <<
        fugitive.name << " and " << sleepless.name << endl;

  cout << "I thought " << fugitive.first_star << " was great!" <<
        endl;
 }
```

După cum se observă, când o funcţie membru este declarată inline, instrucţiunile funcţiei sunt scrise în interiorul clasei:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
  char name[64];
  char first_star[64];
  char second_star[64];
  void show_movie(void)
   {
     cout << "Movie name: " << name << endl;
     cout << "Starring: " << first_star << " and " <<
             second_star << endl << endl;
   }

void initialize(char *movie_name, char *first, char *second)
   {
     strcpy(name, movie_name);
     strcpy(first_star, first);
     strcpy(second_star, second);
   }
};
```


Un avantaj al declarării funcţiilor membru inline este acela că întreaga clasă ocupă un loc compact în program. Din păcate, folosirea funcţiilor inline în acest mod măreşte dimensiunea şi complexitatea definiţiilor clasei. Mai simplu spus, cu cât definiţia clasei devine mai mare, cu atât devine mai dificil de înţeles. În plus, codul pentru funcţiile inline nu este partajat între tipurile de obiecte similare, aşa cum se va vedea în continuare.

Când se definesc funcţiile membru în afara clasei, compilatorul C++ creează o copie a corpului funcţiei ce va fi ulterior folosită de fiecare obiect al acelei clase. Cu alte cuvinte, dacă se creează 1000 de obiecte, fiecare obiect va folosi o singură copie a codului funcţiei. O astfel de partajare a funcţiei este indicată, deoarece reduce semnificativ consumul de memorie al programului.



DEFINIREA FUNCŢIILOR MEMBRU ÎN AFARA CLASEI

Când definiţi funcţiile membru aveţi două posibilităţi. Mai întâi, puteţi defini funcţiile inline în interiorul clasei, astfel încât instrucţiunile funcţiei să apară în definiţia clasei. În al doilea rând, puteţi defini funcţiile în afara clasei. În majoritatea cazurilor, aceasta este varianta optimă, pentru a reduce dimensiunea şi complexitatea clasei şi de a asigura partajarea codului funcţiei între obiecte.

Dacă generaţi listing-ul în limbaj de asamblare al programelor PUBLIC.CPP şi INLINE.CPP, veţi vedea că compilatorul nu va partaja codul funcţiilor declarate inline.

REZOLVAREA CONFLICTELOR DE NUME ÎNTRE MEMBRI ŞI PARAMETRI

Când transferaţi parametri unor funcţii membru ale unei clase, numele parametrilor formali ai funcţiei creează variabile locale:

```
void movie::initialize(char *movie_name, char *first, char *second)
{
  strcpy(name, movie_name);
  strcpy(first_star, first);
  strcpy(second_star, second);
}
```

Nume parametri formali

După cum se observă, această funcţie foloseşte nume diferite pentru parametri şi pentru variabilele membru ale unei clase. Când numele unui parametru intră în conflict cu numele unui membru al unei clase, numele parametrului va fi folosit, iar numele membrului clasei va fi *ascuns*. Să presupunem, de exemplu, că funcţia *initialize* ar folosi următoarele nume de parametri:

```
void movie::initialize(char *name, char *first_star,
  char *second_star)
{
  strcpy(name, name);
  strcpy(first_star, first_star);
  strcpy(second_star, second_star);
}
```

După cum se observă, ar fi imposibil pentru compilatorul C++ să determine care nume aparţin parametrilor şi care aparţin membrilor clasei. De aceea, compilatorul va presupune că fiecare nume aparţine parametrilor.

Deoarece nu putem găsi întotdeauna nume sugestive şi variate pentru parametri şi pentru membrii unei clase, se poate utiliza operatorul de rezoluţie globală (::) din C++, pentru a elimina necesitatea numelor distincte. Când programul

vrea să se refere la un membru al unei clase, numele acestuia trebuie precedat cu numele clasei şi simbolul '::', cum este arătat mai jos:

```
void movie::initialize(char *name, char *first_star,
  char *second_star)
{
  strcpy(movie::name, name);
  strcpy(movie::first_star, first_star);
  strcpy(movie::second_star, second_star);
}
```

Referinţă nume membru

Referinţă nume parametru

Scrierea numelui clasei şi a simbolului :: anterior numelui unui membru al acesteia permite decelarea referinţei la membrul respectiv atât de către compilator, cât şi de alţi programatori care citesc codul.



REZOLVAREA CONFLICTELOR DE NUME

În cadrul funcţiilor membru ale unei clase pot apărea situaţii când numele membrilor clasei poate coincide cu numele parametrilor transmişi funcţiei. În mod prestabilit, C++ rezolvă astfel de conflicte de nume prin folosirea parametrului (variabila locală) şi ascunzând existenţa variabilei membru a clasei. Pentru a preveni astfel de conflicte de nume, scrieţi în faţa numelui membrului clasei numele clasei urmat de simbolul '::', cum se arată mai jos:

```
void dogs::assign_dogs(char *breed, int height, int weight)
{
  strcpy(dogs::breed, breed);
  dogs::height = height;
  dogs::weight = weight;
}
```

În acest caz, numele precedate de *dogs*:: corespund numelor membrilor clasei. Celelalte nume corespund variabilelor locale.

SĂ ÎNŢELEGEM MEMBRII DE TIP PRIVATE AI UNEI CLASE

Fiecare din clasele examinate până acum aveau variabilele şi funcţiile membre declarate de tipul *public*. În acest mod, programul poate avea acces la membrii clasei în orice punct al său (acolo unde clasa este în vigoare), folosind operatorul '.'. Din păcate, accesul la toţi membrii unei clase în acest mod poate duce la erori şi la alte probleme. Iată de ce.

După cum am discutat mai înainte, unul din avantajele folosirii obiectelor este că programatorii nu trebuie să înţeleagă obiectul pentru a-l folosi. Cu alte cuvinte, programatorii pot trata obiectele ca pe o "cutie neagră". Pentru a folosi

un obiect, un programator trebuie să ştie numai scopul obiectului şi câteva funcţii membru. Aşa după cum veţi învăţa, multe obiecte conţin variabile membru ce memorează informaţii importante.

De exemplu, să presupunem că scrieţi un program pentru Pentagon ce controlează toate rachetele nucleare adăpostite în silozuri de pe teritoriul Statelor Unite. În acest caz, programul foloseşte obiectele *silo* într-un mod similar cu cel indicat mai jos:

```
class silo {
  public:
    initialize(int missile_type, char *location);
    bombs_away(char *password);
    int missile_type;
    char location[64];
    int fire_missiles;   // If 0 don't fire, if 1 fire
    char password[64] = "Hillary";
};
```

După cum se observă (din motive de securitate evidente), pentru a lansa o rachetă, programul trebuie să specifice o parolă. Dacă parola este corectă, indicatorul variabilei *fire_missiles* primeşte valoarea 1, iar rachetele lansate. Dacă parola nu e corectă, variabila rămâne 0, şi vom avea pace mondială. Din nefericire, deoarece toţi membrii clasei au fost declaraţi de tip public, programul va avea acces liber la membrii săi. De exemplu, programul va putea folosi următoarea instrucţiune pentru a lansa rachetele, ignorând astfel funcţia cu acces prin parolă *fire_missiles*:

```
wyoming_silo.fire_missiles = 1;
```

Când folosiţi obiecte, accesul la majoritatea variabilelor membru trebuie limitat la funcţiile membru. În acest mod, singura modalitate ca programul să aibă acces la variabila membru *fire_missiles* este prin folosirea unei funcţii membru, adică programul este forţat să joace după regulile impuse de creatorul său.

Pentru a restricţiona accesul la membrii clasei, se pot folosi membri *private*, amplasând eticheta *private* în interiorul declaraţiei de clasă. Următoarea clasă, de exemplu, restricţionează accesul la mai mulţi membri ai clasei *silo*, aplicându-le eticheta *private*:

```
class silo {
  public:
    initialize(int missile_type, char *location);   --- Membri de tip public
    bombs_away(char *password);
  private:
    int missile_type;
    char location[64];
    int fire_missiles; // 0-don't fire, 1-fire         --- Membri de tip private
    char password[64] = "Hillary";
};
```


În acest caz, programul va putea avea acces la funcţiile membru *initialize* şi *bombs-away* folosind operatorul '.', dar variabilele membru vor putea fi folosite numai prin intermediul acestor funcţii. Dacă programul ar încerca să obţină acces direct la variabilele membru, compilatorul C++ va genera erori de sintaxă.

Următorul program, PRIVATE.CPP, modifică clasa *movie* descrisă anterior folosind membri de tip *private*:

```
#include <iostream.h>
#include <string.h>

class movie {
 public:
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
 private:
  char name[64];
  char first_star[64];
  char second_star[64];
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
           endl << endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
   strcpy(name, movie_name);
   strcpy(first_star, first);
   strcpy(second_star, second);
 }

void main(void)
 {
   movie fugitive, sleepless;

   fugitive.initialize("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
     "Meg Ryan");

   fugitive.show_movie();
   sleepless.show_movie();
 }
```

După cum se poate vedea, programul a aplicat eticheta *private* pentru toate cele trei variabile membru. La compilarea şi execuţia acestui program, pe ecran vor fi afişate următoarele:

```
C:\> PRIVATE  <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones

Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan
```

Să observăm că acest program nu mai afişează mesajele programului precedent PUBLIC.CPP:

```
void main(void)
 {
   movie fugitive, sleepless;

   fugitive.initialize("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
     "Meg Ryan");

   cout << "The last two movies I've watched are: " <<
           fugitive.name << " and " << sleepless.name << endl;

   cout << "I thought " << fugitive.first_star << " was great!" <<
           endl;
 }
```

Deoarece variabilele membru nu mai sunt publice, programul nu mai poate avea acces direct la acestea prin operatorul '.'. Dacă se doreşte accesul la aceste variabile, trebuie create funcţii membru special în acest scop. În cazul când credeţi că scrierea unor astfel de funcţii membru e prea dificilă şi preferaţi accesul direct, nu faceţi altceva decât să măriţi posibilitatea apariţiei unor viitoare erori.



SĂ ÎNŢELEGEM MEMBRII DE TIP PRIVATE AI UNEI CLASE

Când declaraţi o clasă, puteţi defini membrii ei ca *public* sau *private* (sau *protected*, cum veţi afla în Capitolul 4). Membrii publici ai unei clase sunt disponibili în tot programul folosind numele obiectului şi operatorul punct (*class. member*) sau operatorul de indirectare (*class_ptr->member*). Membrii *private* ai unei clase, pe de altă parte, pot fi utilizaţi numai prin funcţiile membru ale clasei. Prin declararea într-o clasă a unor membri *private*, programele pot controla mai bine valorile atribuite membrilor clasei şi modul în care aceste valori sunt folosite. În mod prestabilit, toţi membrii unei clase sunt de tip *private*. Membrii publici se pot specifica scriind în faţa lor eticheta *public*. Dacă ulterior se doreşte declararea unor membri *private*, trebuie inserată eticheta *private*, cum se arată în continuare:

```
class some_class {
  public:
    void some_function(char *parameter);              Membri tip public
    void some_other_function(int a, int b, int c);
  private:
    int key_value;                                    Membri tip private
   char password[64];
};
```

În acest caz, funcţiile membru ale clasei sunt publice, în timp ce variabilele membru ale clasei sunt nonpublice. Totuşi, în funcţie de clasă, pot exista funcţii membre nonpublice, sau variabile membru publice. La proiectarea claselor trebuie determinat care dintre membri trebuie să fie de tip public, şi care trebuie să rămână de tip *private*.



SĂ ÎNŢELEGEM CAMUFLAJUL INFORMAŢIEI

Camuflajul informaţiei este procesul de proiectare a funcţiilor sau a claselor drept „cutii negre". Cu alte cuvinte, pentru a folosi o funcţie sau o clasă, programatorul nu trebuie să cunoască toate amănuntele interne de funcţionare ale cutiei, ci mai degrabă operaţia realizată de cutie şi modelul de interfaţare cu cutia. În programele C++, membrii nonpublici ai unei clase realizează camuflajul informaţiei.

SĂ ÎNŢELEGEM CONSTRUCTORII ŞI DESTRUCTORII DE CLASE

Când se creează obiecte, este câteodată nevoie de a aloca memorie pentru zonele tampon (buffers) folosite de obiecte. Să presupunem, de exemplu, că lucraţi cu un obiect *fişier* care memorează numele fişierului într-un şir de caractere. Când creaţi obiectul pentru prima dată, veţi dori ca obiectul să aloce memorie în mod dinamic pentru a memora şirul de caractere. Mai târziu, când obiectul va fi desfiinţat, veţi dori să eliberaţi memoria alocată. Pentru a ajuta programele să realizeze astfel de operaţii de fiecare dată când un obiect este construit şi apoi distrus, limbajul C++ introduce funcţiile de tip *constructor* şi *destructor*. O dată ce v-aţi însuşit aceşti termeni, veţi vedea mai târziu că un constructor nu este altceva decât o funcţie ce se execută automat de fiecare dată când creaţi un obiect, iar un destructor este o funcţie ce se execută automat când obiectul este distrus.

Aţi folosit mai devreme, în acest capitol, funcţia membru *initialize* pentru a iniţializa varibilele membru ale obiectelor aparţinând clasei *movie*. Folosind o funcţie constructor, puteţi elimina necesitatea ca programele dvs. să apeleze funcţii ca *initialize*. În schimb, la crearea obiectului programele vor specifica pur şi simplu valorile parametrilor, ca mai jos:

```
movie fugitive("The Fugitive", "Harrison Ford",
    "Tommy Lee Jones");
  movie.sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");
```

La rândul său, C++ va apela automat funcţia constructor. În cadrul constructorului, puteţi atribui parametrii câmpurilor membru, la fel cum aţi făcut când aţi folosit *initialize*.

O funcţie constructor este specială prin aceea că nu returnează o valoare şi nu are tipul *void*. În al doilea rând, funcţia constructor foloseşte acelaşi nume ca şi clasa. În cazul clasei *movie*, funcţia constructor va fi denumită tot *movie*.

```
class movie {
 public:
  void show_movie(void);
  movie(char *name, char *first, char *second); ——— Funcţii constructor
 private:
  char name[64];
  char first_star[64];
  char second_star[64];
};
```

După cum se poate vedea, funcţia nu specifică o valoare de întoarcere. Ca şi la funcţiile membru ale unei clase, puteţi defini funcţia constructor fie în interiorul clasei (in line), fie în afara acesteia. Următorul program, CONSTRUC.CPP foloseşte funcţia constructor *movie* pentru iniţializarea obiectelor:

```
#include <iostream.h>
#include <string.h>

class movie {
 public:
  void show_movie(void);
  movie(char *name, char *first, char *second);
 private:
  char name[64];
  char first_star[64];
  char second_star[64];
};

void movie::show_movie(void)
 {
```



```
  cout << "Movie name: " << name << endl;
  cout << "Starring: " << first_star << " and " << second_star <<
          endl << endl;
 }                                         Declaraţia funcţiei constructor

movie::movie(char *name, char *first_star, char *second_star)
 {
  strcpy(movie::name, name);
  strcpy(movie::first_star, first_star);
  strcpy(movie::second_star, second_star);
 }

void main(void)
 {
  movie fugitive("The Fugitive", "Harrison Ford",
    "Tommy Lee Jones");
  movie sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");

  fugitive.show_movie();
  sleepless.show_movie();
 }
```

Folosind funcţiile constructor, programele dvs. pot iniţializa obiectele la crearea acestora, eliminând astfel necesitatea unor funcţii de iniţializare separate.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM FUNCŢIILE CONSTRUCTOR

O *funcţie constructor* este o funcţie membru a unei clase pe care C++ o execută automat de fiecare dată când creaţi un obiect al unei anumite clase. Funcţia constructor este specială prin aceea că foloseşte acelaşi nume cu al clasei. De exemplu, pentru o clasă denumită *dogs*, funcţia constructor se va numi tot *dogs*. În plus, funcţia nu returnează nici un rezultat şi nu este de tip *void*.

```
class dogs {
  public:
    dogs(char *breed, int height, int weight); ——— Funcţie constructor
    void show_dogs(void);
  private:
    char breed[64];
    int height;
    int weight;
};
```

SĂ ÎNŢELEGEM FUNCŢIILE DESTRUCTOR

Un destructor este o funcţie ce se execută automat când obiectul este distrus. Ca şi la funcţia constructor, funcţia destructor are acelaşi nume cu clasa. De asemenea, funcţia destructor nu returnează vreo valoare şi nu este de tipul *void*. Programele nu pot transfera parametri la o funcţie destructor. Funcţia destructor se deosebeşte de cea constructor prin faptul că este precedată de semnul tilda (~):

```
class dogs {
   public:
      dogs(char *breed, int height, int weight);
      ~dogs(void); ———————————————— Funcţie destructor
      void show_dogs(void);
   private:
      char breed[64];
      int height;
      int weight;
};
```

Următorul program, DESTRUCT.CPP, adaugă o funcţie destructor la clasa *movie*. Când obiectul este distrus (în acest caz, când programul se încheie), este apelată funcţia destructor:

```
#include <iostream.h>
#include <string.h>

class movie {
 public:
  void show_movie(void);
  movie(char *name, char *first, char *second);
  ~movie(void);
 private:
  char name[64];
  char first_star[64];
  char second_star[64];
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
           endl << endl;
 }

movie::movie(char *name, char *first_star, char *second_star)
 {
   strcpy(movie::name, name);
   strcpy(movie::first_star, first_star);
   strcpy(movie::second_star, second_star);
 }

movie::~movie(void)
 {
   cout << "In the movie destructor for " << name << endl;
 }

void main(void)
 {
   movie fugitive("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   movie sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");

   fugitive.show_movie();
   sleepless.show_movie();
 }
```


La prima vedere, următoarea definiţie de funcţie pare încurcată:

```
movie::~movie(void)
 {
  cout << "In the movie destructor for " << name << endl;
 }
```

Pentru a înţelege antetul funcţiei, să-l examinăm pe porţiuni. Primul *movie*, începând din stânga, urmat de simbolul '::', specifică faptul că această funcţie corespunde clasei *movie*. Apoi, caracterul tilda ('~') ne indică începutul definiţiei funcţiei destructor. Cunoscând acestea, veţi putea deduce că numele *movie* va mai apărea odată deoarece destructorul foloseşte întotdeauna numele clasei. În sfârşit, programul dvs. nu poate transfera parametrii către un destructor, de aceea lista lui de parametri este *void*.

După cum se vede, programul nu apelează în mod explicit destructorul. În schimb, o dată definit un destructor, acesta se execută automat când obiectul este distrus. De obicei, o funcţie destructor va elibera memorie alocată anterior sau va salva informaţii pe un fişier de pe disc. În exemplul anterior, destructorul afişează un mesaj pe ecran, astfel încât la compilarea şi execuţia programului rezultatele afişate vor fi următoarele:

```
C:\> DESTRUCT  <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones

Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan

In the movie destructor for Sleepless in Seattle
In the movie destructor for The Fugitive
```

După cum observaţi, ultimele două linii ale rezultatului sunt scrise de funcţiile destructor.



SĂ ÎNŢELEGEM FUNCŢIILE DESTRUCTOR

Un destructor este o funcţie ce se execută automat când obiectele unei anumite clase sunt distruse. Funcţiile destructor au acelaşi nume ca al clasei în care sunt definite, fiind precedate de semnul tilda (~). Un destructor nu returnează nici o valoare şi nu este *void*. Programul nu poate transmite parametri unui destructor. Următoarea clasă, *employee*, foloseşte o funcţie destructor:

```
class employee {
  public:
    employee(char *name, int age);
    ~employee(void);
    void show_employee(void);
  private:
    char name[64];
    int age;
};
```

Funcţiile destructor, ca şi cele constructor, trebuie să fie membri de tip *public*.

FOLOSIREA FUNCŢIILOR CONSTRUCTOR MULTIPLE

În Capitolul 5 veţi examina suprapunerea funcţiilor, adică folosirea unor funcţii diferite, dar cu acelaşi nume. În timpul compilării, compilatorul C++ determină ce funcţie trebuie apelată, ţinând cont de numărul de parametri sau de tipul valorii funcţiei. Când definiţi funcţii constructor într-un program, puteţi specifica mai multe funcţii, din care compilatorul va selecta funcţia corectă în funcţie de necesităţile programului. De exemplu, următorul program, MULTICON.CPP, foloseşte două funcţii constructor pentru clasa *message*. Primul constructor atribuie mesajul specificat de parametru, în timp ce al doilea constructor foloseşte mesajul prestabilit „Hello, world“:

```
#include <iostream.h>
#include <string.h>

class message {
  public:
    message(char *user_message);
    message(void);
    void show_message(void);
  private:
    char secret_message[64];
```



```
};

message::message(char *user_message)
 {
   strcpy(secret_message, user_message);
 }

message::message(void)
 {
   strcpy(secret_message, "Hello, world");
 }

void message::show_message(void)
 {
   cout << "The message is " << secret_message << endl;
 }

void main(void)
 {
   message greeting;
   message book("Success with C++");

   greeting.show_message();
   book.show_message();
 }
```

Aşa cum se observă, la crearea obiectului *greeting* nu se specifică parametri; în acest caz, programul apelează constructorul *message* care nu are parametri. Când se creează obiectul *book*, programul transmite un şir de caractere constructorului *message* care suportă un parametru de tip şir de caractere. La compilarea şi execuţia acestui program pe ecran vor apărea următoarele:

```
C:\> MULTICON <ENTER>
The message is Hello, world
The message is Success with C++
```

FOLOSIREA ARGUMENTELOR PRESTABILITE LA FUNCŢIILE CONSTRUCTOR

În Capitolul 5 veţi învăţa cum să specificaţi parametrii prestabiliţi pentru funcţii. În cazul apelării unor funcţii fără a specifica valori pentru fiecare parametru, se pot specifica valori prestabilite. Următorul program DEFPARAM.CPP, foloseşte un parametru prestabilit pentru funcţia constructor a clasei *message*:

```
#include <iostream.h>
#include <string.h>

class message {
  public:
    message(char *user_message = "Hello, world");
```

```
    void show_message(void);
  private:
    char secret_message[64];
};

message::message(char *user_message)
 {
   strcpy(secret_message, user_message);
 }

void message::show_message(void)
 {
   cout << "The message is " << secret_message << endl;
 }

void main(void)
 {
   message greeting;
   message book("Success with C++");

   greeting.show_message();
   book.show_message();
 }
```

UN ALT MOD DE A INIŢIALIZA MEMBRII UNEI CLASE

Aşa cum aţi învăţat, funcţiile constructor vă ajută să iniţializaţi membrii unei clase la crearea acesteia. La examinarea programelor C++ se poate întâlni o tehnică de iniţializare mai specială. De exemplu, să presupunem că doriţi să iniţializaţi prin constructorul *counter* variabila *count* la 0, ca mai jos:

```
counter::counter(void)
 {
    counter = 0;
    // Other statements
 }
```

Aşa cum se va dovedi, C++ permite iniţializarea variabilelor membru ale unei clase prin intercalarea numelui variabilei şi a valorii dorite între simbolul ":" şi corpul de instrucţiuni al funcţiei:

```
counter::counter(void) : counter(0)
 {
    // Other statements
 }
```

Numele variabilei şi valoarea iniţială

Următorul program, CON_INIT.CPP foloseşte acest tip de iniţializare pentru atribuirea a trei variabile membru ale clasei *object* valorilor 1, 2 respectiv 3:


```
#include <iostream.h>

class object {
  public:
    object::object(void);
    void show_object(void);
  private:
    int a;
    int b;
    int c;
};
object::object(void) : a(1), b(2), c(3) { };

void object::show_object(void)
 {
   cout << "a contains: " << a << endl;
   cout << "b contains: " << b << endl;
   cout << "c contains: " << c << endl;
 }

void main(void)
 {
   object numbers;

   numbers.show_object();
 }
```

Iniţializarea membrilor clasei

La compilarea şi execuţia acestui program, pe ecran va apărea următorul rezultat:

```
C:\> CON_INIT  <ENTER>
a contains 1
b contains 2
c contains 3
```

ATRIBUIREA VALORII UNUI OBIECT ALTUI OBIECT

Aşa cum într-un program putem atribui valorile unei variabile de tip *int* altei variabile de acelaşi tip, similar putem atribui valorile unui obiect către alt obiect. Veţi vedea că C++ simplifică foarte mult atribuirea de valori obiectelor. De fapt, se foloseşte pur şi simplu operatorul de atribuire (=). De exemplu, următoarea instrucţiune va atribui valoarea unui obiect *date* altui obiect:

```
work_day = today;
```

Presupunând că obiectul *date* conţine membrii *month*, *day* şi *year*, operatorul de atribuire va atribui automat valori tuturor membrilor obiectului. Ca urmare, instrucţiunea anterioară este echivalentă cu instrucţiunile următoare:

```
work_day.month = today.month;
work_day.day = today.day;
work_day.year = today.year;
```

Următorul program, ASSIGN.CPP, foloseşte operatorul de atribuire pentru atribui obiectul *movie* altui obiect:

```
#include <iostream.h>
#include <string.h>

class movie {
 public:
  void show_movie(void);
  void initialize(char *name, char *first, char *second);
 private:
  char name[64];
  char first_star[64];
  char second_star[64];
};

void movie::show_movie(void)
 {
   cout << "Movie name: " << name << endl;
   cout << "Starring: " << first_star << " and " << second_star <<
           endl << endl;
 }

void movie::initialize(char *movie_name, char *first, char *second)
 {
   strcpy(name, movie_name);
   strcpy(first_star, first);
   strcpy(second_star, second);
 }

void main(void)
 {
   movie fugitive, sleepless;
   movie date_choice;

   fugitive.initialize("The Fugitive", "Harrison Ford",
     "Tommy Lee Jones");
   sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
     "Meg Ryan");

   fugitive.show_movie();
   sleepless.show_movie();

   date_choice = sleepless;
   cout << "The date choice is: ";
   date_choice.show_movie();
 }
```


Aşa cum se observă, programul atribuie valoarea obiectului *sleepless* către obiectul *date_choice*. La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> ASSIGN  <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones

Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan

The date choice is: Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan
```

În timpul lucrului cu obiecte, abilitatea de a putea atribui valorile unor obiecte altor obiecte va fi foarte convenabilă. În Capitolul 8 veţi învăţa cum să suprapuneţi operatorii C++. Folosind suprapunerea operatorilor, se va putea testa dacă două obiecte sunt egale printr-o instrucţiune de forma:

```
if (dates_choice == sleepless)
```

În acest caz, suprapunerea operatorului '=' va determina programul să compare valoarea fiecărui membru în parte pentru a determina dacă cele două obiecte sunt egale.

OBIECTE ŞI FUNCŢII

În programe, obiectele şi funcţiile se pot folosi la fel ca structurile şi funcţiile. De exemplu, puteţi transmite unei funcţii un obiect prin valoare sau prin referinţă (în cazul când se doreşte modificarea valorii unui membru). În mod asemănător, funcţiile pot returna obiecte. Următorul program, CLASSFUN.CPP transmite obiectul *message* către două funcţii diferite. Prima funcţie foloseşte apelul prin valoare pentru a afişa variabilele membru ale obiectului. A doua funcţie utilizează apelul prin referinţă pentru a modifica variabilele membru:

```
#include <iostream.h>
#include <string.h>

class message {
  public:
    message(char *user_message, char *owner);
    void show_message(void);
    char message_owner[64];
  private:
    char secret_message[64];
};

message::message(char *user_message, char *owner)
 {
```

```
  strcpy(secret_message, user_message);
  strcpy(message_owner, owner);
}

void message::show_message(void)
{
  cout << "The message owner is " << message_owner << endl;
  cout << "The message is " << secret_message << endl;
}

void some_function(message note)
{
  note.show_message();
}

void change_owner(message *note)
{
  strcpy(note->message_owner, "Fred");
}

void main(void)
{
  message book("Success with C++", "Kris");

  some_function(book);
  change_owner(&book);
  some_function(book);
}
```

După cum se observă, programul transmite obiectul *book* funcţiei *some_function* care, la rândul ei, afişează membrul obiectului folosind funcţia *show_message*. În acest caz, funcţia nu poate afişa valorile variabilelor membru deoarece *secret_message* nu este public. Singurul mod prin care programul poate obţine acces la o variabilă membru este de a folosi o funcţie membru. Mai departe, transmiţând prin referinţă obiectul *book* funcţiei *change_owner*, funcţia va putea modifica membrul *message_owner*. Ca şi înainte, funcţia nu poate schimba membrul *secret_message* deoarece acesta este *private*.



FOLOSIREA OBIECTELOR ŞI A FUNCŢIILOR

Obiectele din programe, ca şi structurile, pot fi transmise funcţiilor drept parametri. De exemplu, dacă o funcţie are nevoie să aibă acces la membrii unei clase, obiectul poate fi transferat funcţiei prin valoare. Dacă funcţia trebuie să modifice un membru al clasei, obiectul trebuie transferat funcţiei prin referinţă. Să nu uităm că o funcţie poate avea acces direct numai la membrii publici ai unei clase. Pentru a modifica un membru nonpublic al clasei, funcţia trebuie să folosească o funcţie membru a clasei.

În Capitolul 10 veţi învăţa cum se transferă clase funcţiilor, folosind referinţe C++, eliminând astfel necesitatea de a aplica operatorul de indirectare (→) în cadrul unei funcţii. În schimb, programele pot avea acces la membrii unei clase folosind operatorul punct (.).

SĂ ÎNŢELEGEM MEMBRII UNEI CLASE

În fiecare din programele anterioare, clasele au folosit tipuri simple de membri, ca *int*, *float* ş.a.m.d. Pe măsură ce clasele devin mai complexe, membrii lor pot fi tablouri, structuri, sau chiar alte clase sau pointeri la clase. De exemplu, următorul program, NESTED.CPP, include un pointer la clasa *date* în interiorul clasei *employee*:

```
#include <iostream.h>
#include <string.h>

class date {
  public:
    date(int month, int day, int year);
    void show_date(void);
  private:
    int month;
    int day;
    int year;
};

class employee {
  public:
    employee(char *name, int age, int month, int day, int year);
    ~employee(void);
    void show_employee(void);
  private:
    char name[64];
    int age;
    date *hire_date;        // Nested class
};

date::date(int month, int day, int year)
{
  date::month = month;
  date::day = day;
  date::year = year;
}

void date::show_date(void)
{
  cout << month << '/' << day << '/' << year << endl;
}
```

```
employee::employee(char *name, int age, int month, int day,
  int year)
 {
   strcpy(employee::name, name);
   employee::age = age;
   hire_date = new date(month, day, year);
 }

employee::~employee(void)
 {
   delete hire_date;
 }

void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
   cout << "Hire date: ";
   hire_date->show_date();
 }

void main(void)
 {
   employee manager("Joe Smith", 33, 12, 25, 1994);

   manager.show_employee();
 }
```

Dacă nu înţelegeţi încă alocarea şi dealocarea memoriei din constructorul şi destructorul clasei *employee*, în Capitolul 7 se va examina alocarea memoriei în detaliu. Deocamdată, reţineţi că membrii unei clase pot fi *tablouri (arrays)*, *structuri (structures)* sau chiar *pointeri* la alte clase.

FOLOSIREA UNUI TABLOU DE CLASE

Aşa cum clasele pot conţine un tablou de valori, programele pot avea tablouri ce conţin mai multe obiecte. De exemplu, următorul program, ARRAY.CPP, foloseşte un tablou ce conţine 5 obiecte de tipul *date*:

```
#include <iostream.h>
#include <string.h>

class date {
  public:
    date(int month, int day, int year);
    date(void);
    ~date(void);
    void show_date(void);
  private:
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
 {
   date::month = month;

   date::day = day;
   date::year = year;
   cout << "In date constructor: ";
   show_date();
 }

date::date(void)
 {
   cout << "In date constructor with no date" << endl;
 }

date::~date(void)
 {
   cout << "In date destructor: ";
   show_date();
 }

void date::show_date(void)
 {
   cout << month << '/' << day << '/' << year << endl;
 }

void main(void)
 {
    date holidays[5];
    date christmas(12, 25, 94);
    date halloween(10, 31, 94);
    date fourth(7, 4, 94);
    date new_years(1, 1, 95);
    date birthday(9, 30, 94);

    holidays[0] = christmas;
    holidays[1] = halloween;
    holidays[2] = fourth;
    holidays[3] = new_years;
    holidays[4] = birthday;
 }
```

După cum se observă, programul iniţializează tabloul în *main*. Când se folosesc tablouri de obiecte, trebuie înţeles că C++ apelează funcţiile constructor şi

destructor pentru fiecare element al tabloului. În acest caz, funcţiile constructor şi destructor afişează un mesaj pentru a vă înştiinţa că au fost apelate. La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> ARRAY  <ENTER>
In date constructor with no date
In date constructor with no date     Constructori pentru
In date constructor with no date     elemente libere din tablou
In date constructor with no date
In date constructor with no date
In date constructor: 12/25/94
In date constructor: 10/31/94        Constructori de date
In date constructor: 7/4/94          individuale
In date constructor: 1/1/95
In date constructor: 9/30/94
In date destructor: 9/30/94
In date destructor: 1/1/95           Destructori pentru
In date destructor: 7/4/94           elemente libere din tablou
In date destructor: 10/31/94
In date destructor: 12/25/94
In date destructor: 9/30/94
In date destructor: 1/1/95           Destructori de date
In date destructor: 7/4/94           individuale
In date destructor: 10/31/94
In date destructor: 12/25/94
```

REZUMAT

În acest capitol au fost prezentate mai multe aspecte ale claselor C++. Următoarele capitole vor examina în detaliu conceptele şi resursele în care stă forţa limbajului C++. Veţi învăţa cum să realizaţi operaţii de intrare/ieşire bazate pe obiecte, cum să folosiţi moştenirea, cum să suprapuneţi operatorii unor clase, cum clasele *template* simplifică declaraţia obiectelor şi multe altele. Înainte de a continua, totuşi, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Un obiect reprezintă o entitate a lumii reale, poate conţine date (variabile membru) şi un set de operaţii (funcţii membru) ce se realizează asupra acestora.
- ✓ Programarea convenţională asimilează programele ca liste lungi de instrucţiuni ce îndeplinesc o anumită sarcină, în timp ce programele orientate pe obiecte se concentrează pe obiectele ce alcătuiesc programul. Dacă vă gândiţi la toate funcţiile unui procesor de texte, de exemplu, veţi dezarma rapid. În schimb, dacă vedeţi procesorul de texte ca pe o colecţie de obiecte distincte, programul va deveni manevrabil.

- ✓ Când scrieţi un obiect (să zicem un obiect fişier) care are anumite caracteristici (tipărire, salvare, redenumire), puteţi refolosi codul acelui obiect în altă secţiune, sau chiar în alt program, economisind astfel timp şi efort.
- ✓ Principalele avantaje ale folosirii obiectelor sunt: uşurinţa în proiectare şi refolosirea codului; fiabilitatea crescută, uşurinţa înţelegerii; abstractizarea sporită; încapsulare şi camuflaj al datelor.
- ✓ Pentru a crea un obiect, se declară variabile de tipul acelui obiect. Pentru a declara o variabilă obiect, se specifică tipul clasei, urmat de numele variabilei obiect.
- ✓ Un avantaj al declarării funcţiilor membre *inline* este că întreaga clasă este localizată într-o porţiune precisă a codului programului. Din nefericire, acest lucru măreşte dimensiunea şi complexitatea definiţiilor de clase, iar codul pentru funcţii *inline* nu este partajat de obiectele similare.
- ✓ Pentru a rezolva conflictul între membrii clasei şi parametrii (variabilele locale) cu acelaşi nume, se foloseşte operatorul de rezoluţie globală (::) la referirea membrului clasei.
- ✓ Un constructor este o funcţie ce se execută automat de fiecare dată când se creează un obiect, eliminându-se nevoia de funcţii de iniţializare separate, iar un destructor este o funcţie ce se execută automat când un obiect este distrus, eliberând de obicei memoria alocată. Ambele funcţii au acelaşi nume ca al clasei, nu returnează valori, şi nu sunt *void*. Destructorul este precedat de tilda (~).
- ✓ Pentru a atribui valoarea unui obiect către alt obiect, se foloseşte operatorul de atribuire (*work_day.year = today.year*).

CAPITOLUL 3
ACOMODAREA CU FIŞIERELE

În Capitolul 1 aţi examinat stream-urile I/O în C++. Aşa cum aţi văzut, compilatorul foloseşte un antet pentru definiţiile claselor de stream-uri I/O, ale metodelor şi ale variabilelor din aceste clase. Folosind manipulatorii şi funcţiile membru, programele pot realiza o varietate de operaţii I/O. Acest capitol examinează operaţiile cu fişiere I/O în detaliu. Veţi învăţa că, folosind stream-urile fişier, programele pot efectua intrări/ieşiri cu operatorii de inserţie sau extracţie şi cu funcţiile membru. La încheierea acestui capitol, veţi înţelege toate operaţiile cu fişiere, inclusiv următoarele:

- Ce trebuie specificat la deschiderea unui fişier
- Ce tipuri de clasă sunt definite în fişierul antet FSTREAM.H
- Cum se deschide un fişier
- Cum se realizează operaţiile de intrare/ieşire cu fişiere, după deschiderea acestora
- Cum se închide un fişier
- Cum se realizează formatarea ieşirii
- Ce este un specificator al modului de deschidere şi controlul modului în care programul deschide fişierul
- Cum se testează reuşita fiecărei operaţii şi modul în care programul poate detecta erorile şi reacţiona la acestea
- Ce sunt operaţiile pe fişiere binare şi cum se scriu sau se citesc informaţiile despre structuri, tablouri sau valori reale
- Operaţii pe fişiere cu acces secvenţial şi acces aleator
- Cum se poziţionează pointerii de fişier pentru operaţii de intrare sau ieşire aleatoare

SĂ ÎNŢELEGEM OPERAŢIILE I/O CU FIŞIERE

În Capitolul 1 s-au dat exemple de programe ce foloseau operaţii I/O cu stream-urile *cin* şi *cout*. Pentru a se folosi aceste stream-uri, în programe se făceau pur şi simplu referinţe la numele acestora. Compilatorul C++ asociază automat ecranul, respectiv tastatura, la cele două stream-uri. Dacă programele citesc informaţii dintr-un fişier, sau scriu informaţii într-un fişier, ele vor trebui, totuşi,

să deschidă fişierul respectiv şi să specifice dacă vor să citească dintr-un fişier de intrare sau să scrie într-un fişier de ieşire.

Pentru a înţelege mai bine stream-urile fişier, ar trebui examinat fişierul antet FSTREAM.H. Veţi putea găsi aici definiţiile claselor *ifstream, ofstream* şi *fstream*. Programele ce folosesc fişiere vor crea obiecte folosind aceste tipuri de clase, în funcţie de natura operaţiei I/O ce trebuie realizată. De exemplu, pentru a realiza operaţii de intrare pe fişiere, veţi folosi clasa *ifstream*. Analog, pentru operaţii de ieşire, se foloseşte clasa *ofstream*. În sfârşit, în cazurile în care programul trebuie să citească sau să scrie în/din acelaşi fişier, veţi folosi clasa *fstream*.

DESCHIDEREA UNUI FIŞIER PENTRU IEŞIRE

Modul cel mai simplu de a deschide un fişier este de a crea un obiect bazat pe una din clasele stream-urilor fişier enumerate anterior. De exemplu, pentru a deschide un fişier pentru ieşire, trebuie mai întâi declarat un obiect de tipul *ofstream*, ca mai jos:

Stream fişier de ieşire

```
ofstream output_file;
```

Nume obiect fişier

Apoi, se poate folosi funcţia membru *open* pentru a deschide fişierul:

Funcţia membru open

```
output_file.open ("FILENAME.EXT");
```

Nume fişier

În acest caz, programul va deschide un fişier de ieşire cu numele FILENAME.EXT. După deschiderea fişierului de ieşire, se poate folosi operatorul de inserţie pentru a scrie în fişier. De exemplu, următoarea instrucţiune va înscrie un mesaj în fişier:

```
output_file <<"Acest mesaj este scris in fisier" <<endl;
```

Când programul nu mai lucrează cu fişierul, trebuie folosită funcţia membru *close* pentru a închide fişierul:

```
output_file.close ( );
```

Următorul program, BOOKINFO.CPP, scrie informaţii despre această carte în fişierul BOOKINFO.DAT:

```
#include <fstream.h>

void main(void)
 {
   ofstream output_file;
```



```
   output_file.open("BOOKINFO.DAT");

   output_file << "Title: Success with C++" << endl;
   output_file << "Author: Kris Jamsa" << endl;
   output_file << "Publisher: Jamsa Press" << endl;
   output_file << "Price: $29.95" << endl;

   output_file.close();
 }
```



DESCHIDEREA UNUI FIŞIER PENTRU IEŞIRE

Pentru a scrie date într-un fişier, programul trebuie să creeze un obiect de tipul *ofstream*, care este definit în fişierul antet FSTREAM.H. Apoi, programul trebuie să deschidă stream-ul folosind funcţia membru *open*. Pentru a scrie date în fişier, se foloseşte operatorul de inserţie. După terminarea operaţiilor, fişierul trebuie închis folosind funcţia membru *close*. Următorul program ilustrează paşii necesari pentru a scrie date într-un fişier:

```
#include <fstream.h>  ———— Include fişierul antet FSTREAM.H

void main(void) ———— Declară un obiect de tip ofstream
 {
   ofstream budget;

   budget.open("BUDGET.DAT");———— Deschide fişierul dorit

   budget << "Payroll Information" << endl; ———— Execută scrierea
   budget << "XYZ Corporation" << endl;

   budget.close(); ——— Închide fişierul
 }
```

FOLOSIREA FUNCŢIEI CONSTRUCTOR PENTRU DESCHIDEREA UNUI FIŞIER

Aşa cum aţi învăţat, programele pot deschide fişiere folosind funcţia membru ***open***. Pe lângă această funcţie, programele pot folosi şi funcţia constructor a clasei pentru a deschide un fişier, cum se indică mai jos:

```
ofstream output_file("FILENAME.EXT");
```

Folosind funcţia constructor în acest fel, se elimină instrucţiunea *open* din program. Ca o regulă generală, reducerea numărului de instrucţiuni într-un program duce la îmbunătăţirea clarităţii programului. De aceea, în majoritatea programelor se va folosi funcţia constructor din *ofstream* pentru deschiderea fişierelor.

FOLOSIREA MANIPULATORILOR ŞI A FUNCŢIILOR MEMBRU DE IEŞIRE

Când se realizează operaţii pe fişiere utilizând stream-uri fişier, se pot folosi manipulatorii de ieşire prezentaţi în Capitolul 1. De exemplu, următorul program, HEXOCTDE.CPP creează un fişier cu numele HEXOCTDE.DAT ce conţine reprezentările în zecimal, octal şi hexazecimal ale valorilor 1 până la 255:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
 {
  int i;

  ofstream output("HEXOCTDE.DAT");

  output << "  Dec " << " Oct " << " Hex " << endl;
  for (i = 0; i <= 255; i++)
    output << dec << setw(5) << i << setw(5) << oct << i <<
              setw(5) << hex << i << endl;

  output.close();
 }
```

Pentru a folosi aceşti manipulatori, trebuie inclus fişierul antet IOMANIP.H, cum s-a arătat la începutul programului anterior.

Pe lângă folosirea manipulatorilor de stream I/O prezentaţi în Capitolul 1, se pot folosi şi funcţiile membru ce corespund stream-urilor I/O. De exemplu, următorul program, ALPHABET.CPP, foloseşte funcţia membru *put* pentru a scrie literele alfabetului în fişierul ALPHABET.DAT:

```
#include <fstream.h>

void main(void)
 {
   ofstream alphabet("ALPHABET.DAT");

   for (int letter = 'A'; letter <= 'Z'; letter++)
     alphabet.put((char)letter);
 }
```

În acest caz, programul foloseşte funcţia membru *put* pentru a scrie valori de tipul *int* sub formă ASCII.

Următorul program, ABC_INS.CPP, realizează o procesare identică, scriind caractere în fişier cu operatorul de inserţie.

```
#include <fstream.h>

void main(void)
 {
   ofstream alphabet("ALPHABET.DAT");
```



```
   for (char letter = 'A'; letter <= 'Z'; letter++)
     alphabet << letter;
 }
```

Să observăm că programul foloseşte variabile de tipul *char*. Dacă ar fi fost folosite variabile de tipul *int*, programul ar fi scris în fişier numerele de la 65 până la 90.

REALIZAREA OPERAŢIILOR DE SCRIERE FORMATATĂ A FIŞIERELOR

Dacă sunteţi familiarizaţi cu C, cunoaşteţi că un fişier poate fi scris cu diverse formatări, folosind funcţia *fprintf*. Când programaţi în C++, puteţi folosi manipulatorii prezentaţi în Capitolul 1 pentru a formata ieşirea. De exemplu, să presupunem că doriţi să creaţi o tabelă de forma:

Name	Age	SSAN	Salary
Robert Jones	51	111-22-3333	$55000.00
Betty Smith	43	333-22-1111	$60000.00
Reggie Allen	30	111-11-0000	$9000.00

Următorul program, TABLE.CPP, foloseşte manipulatorii de formatare pentru crearea tabelei anterioare:

```
#include <fstream.h>
#include <iomanip.h>
void main(void)
 {
  ofstream report("EMPLOYEE.RPT");

  struct {
    char name[64];
    int age;
    char ssan[64];
    float salary;
  } employees[] = {{"Robert Jones", 51, "111-22-3333", 55000.00},
                   {"Betty Smith", 43, "333-22-1111", 60000.00},
                   {"Reggie Allen", 30, "111-11-0000", 9000.00}};

  report << "Name\t\tAge\tSSAN\t\tSalary" << endl;

  for (int i = 0; i < 3; i++)
   {
     report << setiosflags(ios::left) << setw(16) <<
               employees[i].name;
     report << setw(8) << employees[i].age << setw(16) <<
               employees[i].ssan;
     report << setprecision(2) << setiosflags(ios::right |
               ios::showpoint | ios::fixed);
     report << setw(8) << employees[i].salary << endl;
   }

  report.close();
 }
```



SCRIEREA FORMATATĂ A FIŞIERELOR

Pe măsură ce programele devin mai complexe, apare necesitatea realizării unei ieşiri formatate. Folosind manipulatorii de stream I/O prezentaţi în Capitolul 1, ca *setw, setprecision, setiosflags*, programele pot realiza formatarea scrierii fişierelor.

CONTROLUL MODULUI DE DESCHIDERE A FIŞIERELOR DE IEŞIRE

În mod prestabilit, când se deschide un fişier pentru operaţii de ieşire, dacă există deja pe disc un fişier cu acelaşi nume, acesta din urmă va fi suprascris. În funcţie de scopul programului, s-ar putea dori ca un fişier să fie extins prin adăugarea de informaţii în final, sau ca o operaţie pe un fişier să nu se realizeze dacă un fişier cu acelaşi nume deja există pe disc. Pentru a putea controla modul în care sunt deschise fişierele de ieşire, se poate specifica, la deschiderea acestora, un anumit mod de deschidere. De exemplu, următoarea instrucţiune deschide fişierul BUDGET.DAT pentru operaţii de extindere:

```
ofstream output("BUDGET.DAT", ios::app);
```

Dacă fişierul BUDGET.DAT există, programul va deschide fişierul, scriind noi informaţii în finalul acestuia. Dacă fişierul nu există, programul va crea fişierul. Următorul program, LOGFILE.CPP, deschide un fişier denumit LOGFILE.DAT pentru operaţii de extindere. Programul plasează informaţii în fişier, inserând în prealabil data şi ora curentă. De exemplu, dacă doriţi să urmăriţi proiectele la care lucraţi în cursul zilei, puteţi apela LOGFILE astfel:

```
C:\> LOGFILE Formulate budget proposal  <ENTER>
C:\> LOGFILE Start research for new CD-ROM product  <ENTER>
```

În interiorul fişierului LOGFILE.DAT, intrările vor arăta astfel:

```
C:\> TYPE LOGFILE.DAT  <ENTER>
11/09/93 17:07:38 Formulate budget proposal
11/09/93 17:07:50 Start research for new CD-ROM project
```

Următorul program implementează LOGFILE.CPP:

```
#include <fstream.h>
#include <time.h>

void main(int argc, char **argv)
{
   ofstream output_file("LOGFILE.DAT", ios::app);

   char time[9], date[9];
```



```
   if (argc > 1)
   {
     output_file << _strdate(date) << " " << _strtime(time);

     while (*++argv) {
       output_file << " " << *argv;
     } while (*argv);

     output_file << endl;

     output_file.close();
   }
}
```

După cum se poate vedea, programul deschide fişierul folosind specificatorul de mod de extindere *ios::app*. Programul foloseşte funcţiile de bibliotecă *_strdate* şi *_strtime* pentru a obţine data şi ora, după care ciclează prin argumentele liniei de comandă, inserând fiecare argument pe aceeaşi linie cu data şi ora curentă. În final, programul foloseşte funcţia membru *close* pentru a închide fişierul.

CARACTERE SPECIALE ŞI FIŞIERE I/O

În Capitolul 1 aţi folosit caractere speciale ca \n (newline), \t (tab) şi \a (bell ASCII) cu stream-ul de ieşire *cout*. Când programele realizează operaţii de ieşire cu fişiere, se pot folosi toate caracterele speciale prezentate în Capitolul 1. De exemplu, următorul program, HEX_TOO.CPP, schimbă programul anterior HEXOCTDE.CPP pentru a crea o tabelă în hexazecimal, octal şi zecimal ce foloseşte caracterul de tabulare (\t) pentru formatare:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
{
  int i;

  ofstream output("HEXOCTDE.DAT");

  output << "\tDec\tOct\tHex" << endl;
  for (i = 0; i <= 255; i++)
    output << dec << '\t' << i << '\t' << oct << i << '\t' << hex <<
            i << endl;

  output.close();
}
```

ALŢI SPECIFICATORI AI MODULUI DE DESCHIDERE A FIŞIERELOR

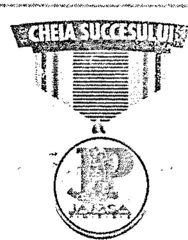
Tocmai aţi învăţat că specificatorul de mod *ios::app* permite unui program să deschidă un fişier în modul extindere. Aşa cum se arată în Tabela 3.1. funcţia membru *open* suportă şi alţi specificatori de deschidere.

Modul de deschidere	Ce se realizează
ios::app	Deschide un fişier de ieşire pentru operaţii de extindere
ios::ate	Deschide un fişier pentru intrare sau ieşire, plasând pointerul de fişier la sfârşitul fişierului
ios::in	Deschide un fişier pentru intrare. Acest mod este prestabilit pentru obiecte de tip ifstream
ios::out	Deschide un fişier pentru ieşire. Acest mod este prestabilit pentru obiectele de tip *ofstream*
ios::nocreate	Deschide un fişier presupus existent. Dacă fişierul nu există, operaţia de deschidere nu se realizează
ios::noreplace	Creează un fişier nou. Dacă un fişier cu acelaşi nume există deja, operaţia de deschidere nu se realizează
ios::trunc	Şterge un fişier existent şi creează un nou fişier
ios::binary	Deschide un fişier pentru operaţii de citire şi scriere binară

Tabela 3.1. Modurile de deschidere a fişierelor, aşa cum sunt definite în fişierul antet folosit la definirea stream-urilor.

Dacă examinaţi conţinutul fişierului antet folosit de compilator la definirea stream-urilor, veţi găsi un tip enumerativ ce defineşte modurile de deschidere a fişierelor, de forma următoare:

```
enum open mode {
in = 0 x 01,        // Deschide un fisier pentru intrare
out = 0 x 02,       // Deschide un fisier pentru iesire
ate = 0 x 04,       // Pozitioneaza pointerul de fisier
                           // la sfarsitul fisierului
app = 0 x 08,       // Pozitioneaza pointerul unui fisier de iesire
                           // la sfarsitul fisierului
trunc = 0 x 10,    // Trunchiaza un fisier existent
nocreate = 0 x 20, // Deschide un fis. existent sau nu deschide
noreplace = 0 x 40, // Deschide un fis. nou sau op. nu se realizeaza
binary = 0 x 80     // Deschide un fisier in mod binar
};
```



CONTROLUL OPERAŢIILOR PE FIŞIERE

Când un program prelucrează fişiere de ieşire, apar situaţii când informaţiile trebuie scrise în continuarea unui fişier existent, sau când conţinutul unui fişier nu trebuie şters. În astfel de situaţii, programele pot folosi specificatorii de mod în cadrul operaţiei de deschidere. De exemplu, următoarea instrucţiune deschide un fişier pentru operaţii de extindere:


```
ofstream output("FILENAME.EXT", ios::app);
```

Analog, următoarea instrucţiune foloseşte indicatorul *ios::noreplace* pentru a preveni suprascrierea unui fişier existent:

```
ofstream output("FILENAME.EXT", ios::nopreplace);
```

Folosind specificatorii modului de deschidere, programele pot prelua controlul asupra operaţiilor de deschidere a fişierelor.

DESCHIDEREA FIŞIERELOR PENTRU OPERAŢII DE INTRARE

Pentru a deschide un fişier pentru operaţii de intrare, programele creează, în mod normal, un obiect de tipul *ifstream*. Ca şi în cazul când deschideţi un fişier pentru ieşire, la deschiderea unui fişier pentru intrare se poate folosi atât funcţia *open*, cât şi funcţia constructor a clasei *ifstream*. Prin urmare, fiecare din următoarele secvenţe are ca efect deschiderea unui fişier denumit BUDGET.DAT:

```
ifstream input_file;
input_file.open("BUDGET.DAT");
```

```
ifstream input_file("BUDGET.DAT");
```

După ce un fişier este deschis pentru intrare, programul poate citi informaţii folosind operatorul de extracţie sau funcţiile membru, cum se se va arăta în continuare.

REALIZAREA OPERAŢIILOR DE INTRARE PE FIŞIERE

După deschiderea unui fişier de intrare, conţinutul său poate fi citit folosind funcţiile membru ale clasei *ifstream* sau operatorul de extracţie. La citirea unui fişier, informaţiile vor fi citite succesiv, de la începutul fişierului, până când va fi întâlnit marcajul de sfârşit de fişier. Pentru a determina dacă s-a ajuns la sfârşitul de fişier, programele pot folosi funcţia membru *eof* într-un ciclu *while*, ca mai jos:

```
while (! input_file.eof())
 {
   // Read and process the file
 }
```

Următorul program, SHOWFILE.CPP, deschide pentru intrare un fişier specificat ca argument în linia de comandă, apoi citeşte şi afişează conţinutul său:

```
#include <fstream.h>

void main(int argc, char **argv)
 {
   ifstream input(argv[1]);

   if (input.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
```

```
    {
      while (! input.eof())
        cout.put((char)input.get());

      input.close();
    }
}
```

***Observaţie:** Operatorul de mulare a tipurilor,* (char)*, necesar în programul anterior pentru a evita un apel ambiguu al funcţiei* put()*, a fost înlăturat din exemplele ulterioare pentru a îmbunătăţi claritatea acestora.*

După cum se observă, programul foloseşte următorul ciclu pentru a afişa conţinutul fişierului, literă cu literă:

```
while (! input.eof())
  cout.put(input.get());
```

Ciclul combină, de fapt, două instrucţiuni de citire şi afişare a câte unui caracter:

```
letter = input.get();    // Read a character

cout.put(letter);        // Output the character
```

La întâlnirea sfâşitului de fişier, ciclul se termină, iar programul închide fişierul.

Pentru a afişa conţinutul fişierului, programul SHOWFILE va fi apelat cu numele fişierului respectiv, ca mai jos:

```
C:\> SHOWFILE  FILENAME.EXT  <ENTER>
```

Dacă fişierul specificat nu poate fi deschis, programul detectează eroarea folosind funcţia membru *fail*:

```
if (input.fail())
  cerr << "Error opening the file: " << argv[1] << endl;
```

În mod similar, programul MOREFILE.CPP citeşte şi afişează conţinutul fişierului, pagină cu pagină. După afişarea unei pagini de ecran, programul este pus în aşteptare, până când utilizatorul apasă o tastă:

```
#include <fstream.h>

void main(int argc, char **argv)
 {
   char line[128];

   long line_count = 0L;

   ifstream input(argv[1]);

   if (input.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
```



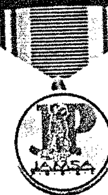
```
   {
     while (! input.eof())
       {
         input.getline(line, sizeof(line));
         cout << line << endl;

         if ((++line_count % 24) == 0)
           {
             cout << "-MOREFILE-";
             cin.get();
           }
       }
     input.close();
   }
 }
```

După cum se poate vedea, programul deschide fişierul specificat pentru operaţii de intrare, folosind primul argument al liniei de comandă. Dacă are loc o eroare, programul afişează un mesaj de eroare şi se încheie. În caz contrar, programul ciclează, citind conţinutul fişierului prin funcţia membru *getline*, până când este întâlnit sfârşitul de fişier. După cum se observă, programul foloseşte funcţia membru *eof* pentru a detecta sfârşitul de fişier. După afişarea a 24 de linii de text, programul rămâne în aşteptare până la apăsarea unei taste. Programul foloseşte operatorul % pentru a determina umplerea ecranului cu 24 de linii. De fiecare dată când numărul liniei curente împărţit la 24 dă restul 0, deja 24 de linii sunt afişate pe ecran, iar programul invită utilizatorul să apese o tastă.

Pentru a afişa conţinutul fişierului FILENAME.EXT pagină cu pagină, trebuie apelat programul MOREFILE astfel:

```
C:\> MOREFILE  <  FILENAME.EXT  <ENTER>
```

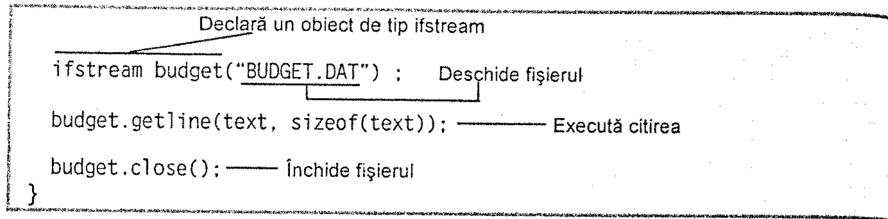


DESCHIDEREA UNUI FIŞIER PENTRU INTRARE

Pentru a citi informaţii dintr-un fişier, programul trebuie să creeze un obiect de tipul *ifstream*, definit în fişierul antet FSTREAM.H. Apoi, fişierul trebuie deschis prin funcţia membru *open* sau prin funcţia constructor. În continuare, programul poate folosi operatorul de extracţie pentru a citi date din fişier. După terminarea prelucrării, fişierul trebuie închis folosind funcţia membru *close*. Următorul program ilustrează paşii ce trebuie urmaţi pentru a citi date dintr-un fişier:

```
#include <fstream.h>  ——— Include fişierul antet FSTREAM.H

void main(void)
 {
   char text[256];
```



TESTAREA REUŞITEI UNEI OPERAŢII I/O

Două din programele anterioare au folosit funcţia membru *fail* pentru a determina dacă programul a putut deschide cu succes un fişier pentru intrare, cum se arată mai jos:

```
ifstream input(argv[1]);

if (input.fail())
  cerr << "Error opening the file: " << argv[1] << endl;
```

Aşa cum aţi învăţat la Capitolul 1, funcţia membru *fail* returnează valoarea 1 dacă a avut loc o eroare la ultima operaţie pe fişier. În acest caz, programul a detectat numai erorile la deschiderea fişierului. În Capitolul 1 aţi învăţat totuşi că trebuie să testaţi reuşita fiecărei operaţii de citire sau scriere. Următorul program, ASCICOPY.CPP, citeşte conţinutul primului fişier specificat în linia de comandă, copiind conţinutul acestuia în al doilea fişier al liniei de comandă.

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   ifstream source(argv[1]);

   char line[128];

   if (source.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
    {
      ofstream target(argv[2]);

      if (target.fail())
        cerr << "Error opening the file: " << argv[2] << endl;
      else
        {
          while (! source.eof())
            {
              source.getline(line, sizeof(line));

              if (source.good())
                {
                  target << line << endl;
                  if (target.fail())
                    {
                      cerr << "Error writing the file: " <<
                              argv[2] << endl;
                      exit(1);
                    }
                }
              else if (! source.eof())
                {
                  cerr << "Error reading the file: " <<
                          argv[1] << endl;
                  exit(1);
                }
            }
          source.close();
          target.close();
        }
    }
 }
```


După cum se observă, programul foloseşte funcţiile membru *good* şi *fail* pentru a testa dacă fişierele au fost deschise cu succes şi dacă operaţiile de citire şi scriere au reuşit. La apariţia unei erori, programul afişează un mesaj de eroare şi foloseşte funcţia *exit* din biblioteca sistemului pentru a se încheia. Aşa cum se vede, programul testează mai întâi dacă operaţia de citire a reuşit. În caz afirmativ, programul scrie datele în fişierul destinaţie.

După cum se observă, programul foloseşte funcţia membru *good* pentru a testa reuşita unei operaţii de citire. Aşa cum se dovedeşte, când se întâlneşte sfârşitul de fişier la citire, indicatorul *fail* este activ. Dacă programul nu testează atingerea sfârşitului de fişier în timpul detecţiei erorilor, atunci el ar putea trata finalul de fişier (care este un eveniment reuşit) drept o eroare. Folosind funcţia membru *good*, programul elimină această eroare posibilă.

Observaţie: *Programul ASCICOPY.CPP va copia numai fişiere text ASCII, cum sunt fişierele programelor sursă C++. Dacă încercaţi să copiaţi un fişier binar, ca de exemplu un program executabil, operaţia de copiere nu se va realiza. Pentru a copia un fişier binar, trebuie să deschideţi fişierul în mod binar, cum se va arăta în continuare.*

Programele precedente au folosit funcţia membru *fail* pentru a determina reuşita sau eşecul unor operaţii I/O pe fişiere. Aşa cum aţi citit în Capitolul 1, programele pot folosi funcţiile membru din Tabela 3.2 pentru determinarea erorilor I/O pe fişiere:

Funcţia membru	Destinaţia
good	Returnează 1 dacă operaţia precedentă a reuşit
eof	Returnează 1 dacă se întâlneşte sfârşitul de fişier
fail	Returnează 1 dacă a apărut o eroare
bad	Returnează 1 dacă s-a efectuat o operaţie incorectă

Tabela 3.2. Funcţiile membru ale stream-ului fişier pentru testarea erorilor.

Aşa cum aţi învăţat în Capitolul 1, programele pot testa reuşita unei operaţii I/O pe stream-uri folosind operatorul de exclamare (!). Astfel, următoarele instrucţiuni sunt identice:

```
if (input.fail())                    if (! input)
```

În mod similar, următoarele două instrucţiuni testează ambele reuşita unei operaţii I/O:

```
if  (input.good())                   if  (input)
```

REALIZAREA OPERAŢIILOR PE FIŞIERE BINARE

În mod prestabilit, operaţiile pe fişiere sunt realizate în mod text. În funcţie de destinaţia programelor, există situaţii când trebuie realizate operaţii pe fişiere binare. De exemplu, aţi creat anterior programul ASCICOPY.CPP, care copia conţinutul unui fişier în alt fişier. Aşa cum s-a spus, acel program nu putea copia fişiere binare, cum sunt programele executabile. Şi iată motivul:

Când un program citeşte un fişier în mod text, programul consideră valoarea ASCII 26 (CTRL-Z) drept sfârşit de fişier. Dacă programul încearcă să citească un fişier binar, este probabil ca programul să întâlnească această valoare undeva, în mijlocul fişierului, determinând ca funcţia *eof* să ia valoarea *true*. Pentru a rezolva această problemă, fişierul trebuie deschis în mod binar, ca mai jos:

```
ifstream input(argv[1], ios::binary);
```

Pentru a realiza operaţii de intrare şi ieşire, programele trebuie să folosească funcţiile membru *read* şi *write*.

```
input.read(buffer, sizeof(buffer));

output.write(buffer, sizeof(buffer));
```

De exemplu, să presupunem că programul trebuie să scrie într-un fişier 30 de preţuri ale unor mărfuri. Următorul program, WRTSTOCK.CPP scrie valori reale în fişierul STOCKS.DAT folosind operaţii pe fişiere binare:

```
#include <fstream.h>

void main(void)
 {
   int count;

   float price;

   ofstream stocks("STOCKS.DAT", ios::binary);

   if (stocks.fail())
     cerr << "Error opening the file STOCKS.DAT" << endl;
   else
     {
       for (count = 1; count <= 30; count++)
        {
          price = count * 100.0;

          stocks.write((char *) &price, sizeof(float));
        }

       stocks.close();
     }
 }
```

Se observă că programul apelează funcţia membru *write*, transferând funcţiei adresa valorii pentru a fi scrisă în fişier:

```
stocks.write((char *) &price, sizeof(float));
```

Dacă încercaţi să afişaţi conţinutul fişierului STOCKS.DAT folosind comanda TYPE, operaţia va eşua, iar pe ecran vor apare caractere neinteligibile. Amintiţi-vă că fişierul STOCKS.DAT este fişier binar, nu fişier ASCII. Pentru a afişa preţurile mărfurilor se poate folosi programul RDSTOCK.CPP, prezentat în continuare:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
 {
   float price;

   ifstream stocks("STOCKS.DAT", ios::binary);

   while (! stocks.eof())
     {
       stocks.read((char *) &price, sizeof(float));

       cout << setprecision(2) << setiosflags(ios::showpoint |
               ios::fixed) << price << endl;
     }

   stocks.close();
 }
```

După cum se vede, programul ciclează, citind valorile reale din fişier, până la întâlnirea marcajului de sfârşit de fişier.

Aţi creat mai devreme programul ASCICOPY.CPP, care copia conţinutul unui fişier ASCII în alt fişier. Următorul program, BIN_COPY.CPP, deschide fişierele sursă şi destinaţie în mod binar, ceea ce permite programului să copieze atât fişiere text, cât şi fişiere binare:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   ifstream source(argv[1], ios::binary);

   char line[1];

   if (source.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
    {
      ofstream target(argv[2], ios::binary);

      if (target.fail())
        cerr << "Error opening the file: " << argv[2] << endl;
      else
        {
          while (! source.eof() && ! source.fail())
            {
              source.read(line, sizeof(line));

              if (source.good())
               {
                 target.write(line, sizeof(line));

                 if (target.fail())
                   {
                     cerr << "Error writing the file: " <<
                             argv[2] << endl;
                     exit(1);
                   }
               }
              else if (! source.eof())
               {
                 cerr << "Error reading the file: " <<
                         argv[1] << endl;
                 exit(1);
               }
            }
          source.close();
          target.close();
        }
    }
 }
```

După cum se observă, programul deschide ambele fişiere pentru operaţii binare folosind specificatorul de mod *ios::binary*. Programul foloseşte funcţia membru *read* pentru a citi date de la intrare şi funcţia membru *write* pentru a scrie date pe fişierul de ieşire. În plus, programul foloseşte funcţiile membru *fail* şi *good* pentru a determina dacă operaţiile I/O se termină cu succes. La întâlnirea marcajului la sfârşit de fişier, programul închide ambele fişiere.



REALIZAREA OPERAŢIILOR PE FIŞIERE BINARE

Pe măsură ce informaţia memorată de programe devine mai complexă, apar situaţii când programele trebuie să citească şi să scrie informaţii binare (ca de exemplu date de tip real sau structuri). Pentru a realiza operaţii pe un fişier binar, trebuie mai întâi deschis fişierul folosind specificatorul *ios::binary*. De exemplu, următoarea instrucţiune deschide un fişier de ieşire numit OUTPUT.DAT pentru operaţii binare:

```
ofstream data ("OUTPUT.DAT", ios::binary);
```

Pentru a realiza intrări/ieşiri în binar, programele vor folosi funcţiile membru *read* şi *write*, specificând adresa de început a zonei de date precum şi numărul de octeţi ce urmează a fi introduşi sau extraşi:

```
input_file.read (& data, sizeof(data));

output_file.write (& data, sizeof(data));
```

După terminarea operaţiilor de intrare/ieşire, programele trebuie să închidă fişierul folosind funcţia membru *close*.

ATENŢIE LA OPERAŢIILE DE INSERŢIE/EXTRACŢIE CU FIŞIERE BINARE

Aşa cum aţi văzut, operaţiile de intrare/ieşire pe fişiere binare se efectuează prin funcţiile membru *read* şi *write*. Dacă se folosesc operatorii de inserţie/extracţie pentru operaţii I/O pe fişiere binare, puteţi întâlni erori neaşteptate. Să considerăm, de exemplu, programul BAD_OUT.CPP, care scrie câteva valori reale în fişierul BAD_DATA.DAT folosind operatorul de inserţie:

```
#include <fstream.h>

void main(void)
 {
   ofstream output("BAD_DATA.DAT", ios::binary);

   output << 100.0 / 11.1;
   output << 22.0 / 7.0;
   output << 100.0 / 11.1;

   output.close();
 }
```

Următorul program, BAD_IN.CPP, citeşte, la rândul lui, conţinutul fişierului BAD_DATA.DAT, afişând valorile citite şi valorile pe care fişierul ar trebui să le conţină:

```
#include <fstream.h>

void main(void)
 {
   ifstream input("BAD_DATA.DAT", ios::binary);

   float value;

   input >>  value;
   cout << value << " should be " << 100.0 / 11.1 << endl;
   input >> value;
   cout << value << " should be " << 22.0 / 7.0 << endl;
   input >> value;
   cout << value << " should be " << 100.0 / 11.1 << endl;

   input.close();
 }
```

La compilarea şi execuţia acestui program, pe ecran vor apărea rezultate similare cu următoarele:

```
C:\> BAD_IN  <ENTER>
9.009001 should be 9.009001
0.142869 should be 3.14286
0.00901 should be 9.009001
```

După cum se observă, valorile citite din fişier nu corespund valorilor scrise iniţial în fişier. Şi iată de ce. În mod prestabilit, când programele efectuează ieşiri pe flux, funcţiile stream-ului de ieşire convertesc valorile de diferite tipuri în reprezentări ASCII. De aceea, dacă am dori să vizualizăm conţinutul fişierului BAD_DATA.DAT folosind comanda TYPE, pe ecran ar apărea:

```
C:\> TYPE  BAD_DATA  <ENTER>
9.0090010.142869.009001

9.009001   3.14286   9.009001
```

Când un program va citi mai târziu valorile din acest fişier, el va încerca să convertească caracterele ASCII în valori reale. Din nefericire, aşa cum s-a văzut la execuţia programului, această conversie nu se realizează corect.

Dacă totuşi doriţi să folosiţi operatorii de inserţie/extracţie cu fişierele binare, trebuie să suprapuneţi aceşti operatori pentru a folosi funcţiile *read* şi *write* descrise anterior. Din păcate, C++ va permite suprapunerea operatorilor de inserţie/extracţie numai pentru tipuri de date noi, definite în program. Următorul

program, BIN_OPS.CPP, suprapune operatorul de inserţie pentru valori de tipul *hidefloat*. În acest fel programul reuşeşte să prelucreze fişierele binare folosind operatorii de inserţie/extracţie:

```
#include <fstream.h>

struct hidefloat { float data; };

ostream& operator <<(ostream& file, hidefloat value)
 {
   file.write((char *) &value.data, sizeof(float));
   return(file);
 }

void main(void)
 {
   ofstream output("BAD_DATA.DAT", ios::binary);

   hidefloat value;

   value.data = 100.0 / 11.1;
   output << value;

   value.data = 22.0 / 7.0;
   output << value;

   value.data = 100.0 / 11.1;
   output << value;

   output.close();
 }
```

De fiecare dată când compilatorul C++ întâlneşte o valoare de tipul *hidefloat* asociată cu operatorul de inserţie şi cu un stream fişier de ieşire, programul va apela funcţia operator ce foloseşte, la rândul ei, funcţia membru *write* pentru a scrie valoarea binară în fişier.

Pentru citirea unui fişier binar, se poate folosi următorul program, READ_BIN.CPP, arătat mai jos:

```
#include <fstream.h>

struct hidefloat { float data; };

ifstream& operator >>(ifstream& file, hidefloat *value)
 {
   file.read((char *) value, sizeof(float));
   return(file);
 }

void main(void)
```

```
{
  ifstream input("BAD_DATA.DAT", ios::binary);

  hidefloat value;

  input >> &value;
  cout << value.data << " should be " << 100.0 / 11.1 << endl;
  input >> &value;
  cout << value.data << " should be " << 22.0 / 7.0 << endl;
  input >> &value;
  cout << value.data << " should be " << 100.0 / 11.1 << endl;

  input.close();
}
```

La întâlnirea unui pointer către o variabilă de tipul *hidefloat* asociată cu operatorul de extracţie şi un stream fişier de intrare, programul va apela funcţia operatorului. Deoarece funcţia schimbă valoarea membrului unei structuri, ea primeşte adresa structurii. Aşa cum vedeţi, folosirea operatorilor de inserţie/extracţie este posibilă şi la fişiere binare, dar devine destul de încurcată şi dificil de înţeles.

DESCHIDEREA FIŞIERELOR PENTRU OPERAŢII DE CITIRE ŞI SCRIERE

Fiecare din programele anterioare a deschis fişiere numai pentru intrare sau numai pentru ieşire, dar nu pentru ambele tipuri de operaţii. Când se creează fişiere pentru baze de date sau alte operaţii bazate pe fişiere, apar situaţii când acelaşi fişier trebuie deschis atât pentru operaţii de intrare, cât şi pentru operaţii de ieşire. În astfel de situaţii, trebuie creat un obiect fişier de tipul *fstream*, ca mai jos:

```
fstream database("DATABASE.DAT", ios::in | ios::out);
```

La deschiderea unui fişier atât pentru operaţii de intrare, cât şi de ieşire, programul lucrează cu doi pointeri de fişier - unul pentru intrare şi altul pentru ieşire. În multe situaţii, fişierele deschise pentru intrare şi pentru ieşire sunt folosite pentru acces aleator, cum se va arăta în continuare.

FIŞIERE CU ACCES ALEATOR

Toate programele prezentate până acum în acest capitol au realizat operaţii *secvenţiale* pe fişiere, prin parcurgerea fişierului de la începutul acestuia spre capătul său. Operaţiile cu *acces aleator*, pe de altă parte, nu pornesc obligatoriu de la începutul fişierului. În schimb, ele pot trece de la o locaţie la alta a fişierului, în orice ordine. Dacă s-a deschis un fişier atât pentru intrare, cât şi pentru ieşire, programul poate deplasa pointerul de citire la o locaţie a fişierului, iar pointerul de scriere la altă locaţie. Pentru a deplasa pointerul fişierului de la o locaţie la alta a fişierului, un program poate folosi funcţiile membru *seekg* şi *seekp*. Funcţia *seekg* poziţionează pointerul de intrare al fişierului (denumirea *seekg* derivă din cuvintele *seek* şi *get* adică poziţionare în vederea unei operaţii

de intrare pe fişier). Asemănător, funcţia *seekp* poziţionează pointerul fişierului pentru efectuarea unei operaţii de ieşire (*seek* şi *put*). Formatele funcţiilor *seekg* şi *seekp* sunt următoarele:

```
seekp(offset, from_position);
seekg(offset, from_position);
```

Parametrul *offset* indică un deplasament în octeţi în cadrul fişierului, şi poate fi un număr întreg, pozitiv sau negativ. Parametrul *from_position* specifică locaţia din fişier în raport cu care se realizează deplasamentul. Tabela 3.3 indică valorile enumerative ce pot fi specificate pentru parametrul *from_position*.

Valori enumerative	Poziţia fişierului
ios::beg	De la începutul fişierului
ios::cur	De la poziţia curentă a pointerului în fişier
ios::end	De la sfârşitul fişierului

Tabela 3.3. *Valorile enumerative care specifică poziţiile de bază într-un fişier cu acces aleator.*

Următoarea instrucţiune, de exemplu, deplasează pointerul de ieşire (*put*) al fişierului la sfârşitul fişierului:

```
data_file.seekp(0, ios::end);
```

Pentru a înţelege mai bine operaţiile pe fişiere cu acces aleator, să analizăm programul ABC.CPP, care creează fişierul LETTERS.DAT ce conţine literele A până la H, urmate de literele O până la Z:

```
#include <fstream.h>

void main(void)
{
  ofstream output("LETTERS.DAT");

  for (char letter = 'A'; letter <= 'H'; letter++)
    output << letter;

  for (letter = 'O'; letter <= 'Z'; letter++)
    output << letter;

  output.close();
}
```

Folosiţi comanda TYPE pentru a afişa conţinutul fişierului, ca mai jos:

```
C:\> TYPE  LETTERS.DAT  <ENTER>
ABCDEFGHOPQRSTUVWXYZ
```

În continuare, programul RANDOM.CPP deschide fişierul LETTERS.DAT pentru operaţii de citire şi scriere. Pentru început, programul poziţionează pointerul de scriere al fişierului pe poziţia octetului ce urmează literei H din fişier. Programul

scrie apoi în fişier literele de la I până la Z. În continuare, pointerul de citire al fişierului este poziţionat la începutul acestuia, după care programul citeşte şi afişează conţinutul fişierului:

```
#include <fstream.h>

void main(void)
 {
   fstream letters("LETTERS.DAT", ios::in | ios::out);

   letters.seekp(8, ios::beg);

   for (char letter = 'I'; letter <= 'Z'; letter++)
     letters << letter;

   letters.seekg(0, ios::beg);

   while (! letters.eof())
     cout.put((char)letters.get());

   letters.close();
 }
```

La utilizarea fişierelor cu acces aleator, pot apărea situaţii când este nevoie de a determina poziţia curentă a pointerilor de citire şi scriere ai fişierului. Pentru a obţine aceste valori, se folosesc funcţiile membru tellg şi tellp, cum este arătat mai jos:

```
input_position = file_object.tellg();

output_position = file_object.tellp();
```

RECAPITULAREA MODURILOR DE DESCHIDERE A UNUI FIŞIER

Aşa cum s-a văzut, programele deschid fişierele folosind specificatorii modului de deschidere a fişierelor. De exemplu, un fişier poate fi deschis în modul extensie folosind specificatorul *ios::app*. Următorul program, ASCIAPPD.CPP, adaugă conţinutul primului fişier specificat în linia de comandă la sfârşitul celui de al doilea fişier:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   ifstream source(argv[1]);

   char line[128];

   if (source.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
```



```
   {
     ofstream target(argv[2], ios::app);

     if (target.fail())
       cerr << "Error opening the file: " << argv[2] << endl;
     else
       {
         while (! source.eof())
           {
             source.getline(line, sizeof(line));

             if (source.good())
               {
                 target << line << endl;

                 if (target.fail())
                   {
                     cerr << "Error writing the file: " <<
                             argv[2] << endl;
                     exit(1);
                   }
               }
             else if (! source.eof())
               {
                 cerr << "Error reading the file: " <<
                         argv[1] << endl;
                 exit(1);
               }
           }
         source.close();
         target.close();
       }
   }
 }
```

După cum se observă, programul deschide fişierul de ieşire pentru operaţii de extindere folosind modul *ios::app*. Programul realizează apoi operaţii de citire/scriere standard, folosind funcţiile membru *fail* şi *good* pentru a verifica reuşita fiecărei operaţii. Pentru a extinde fişierul WEEKLY.NTS cu conţinutul fişierului TODAY.NTS se poate apela programul ASCIAPPD astfel:

```
C:\> ASCIAPPD  TODAY.NTS  WEEKLY.NTS  <ENTER>
```

Tot în acest capitol, aţi creat mai înainte fişierul ASCICOPY.CPP care copiază conţinutul unui fişier ASCII în alt fişier. Dacă al doilea fişier există deja, programul va suprascrie conţinutul fişierului existent. În alte situaţii nu este recomandabilă o astfel de suprascriere (deci o distrugere) a fişierelor. Pentru a preveni suprascrierea fişierelor, se poate folosi specificatorul de mod de deschidere *ios::noreplace*, cum se arată mai jos:

```
ofstream  output_file("FILENAME.EXT", ios::noreplace);
```

Următorul program, OVERWRIT.CPP, foloseşte specificatorul *ios::noreplace* pentru a preveni distrugerea unui fişier printr-o operaţie de copiere.

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   ifstream source(argv[1], ios::binary);

   char line[1];

   if (source.fail())
     cerr << "Error opening the file: " << argv[1] << endl;
   else
    {
      ofstream target(argv[2], ios::binary | ios::noreplace);

      if (target.fail())
        cerr << "Error opening the file: " << argv[2] <<
                " or file exists" << endl;
      else
        {
          while (! source.eof() && ! source.fail())
            {
              source.read(line, sizeof(line));

              if (source.good())
               {
                 target.write(line, sizeof(line));

                 if (target.fail())
                   {
                     cerr << "Error writing the file: " <<
                             argv[2] << endl;
                     exit(1);
                   }
               }
              else if (! source.eof())
               {
                 cerr << "Error reading the file: " <<
                         argv[1] << endl;
                 exit(1);
               }

            }
          source.close();
          target.close();
        }
    }
 }
```


După cum se observă, dacă fişierul destinaţie există, programul va afişa un mesaj de eroare şi se va încheia. Programul poate fi modificat de aşa natură încât să întrebe utilizatorul dacă un fişier existent poate sau nu să fie suprascris.



SĂ ÎNŢELEGEM FIŞIERELE CU ACCES ALEATOR

Majoritatea programelor realizează intrări/ieşiri secvenţiale, prelucrând fişierele de la început spre sfârşit. Operaţiile pe fişiere cu acces aleator, pe de altă parte, permit programelor să efectueze citiri sau scrieri de pe orice locaţie a fişierului. Înainte de a începe operaţiile, programul poziţionează pointerul de fişier pe locaţia dorită folosind funcţiile membru *seekp* şi *seekg*. Funcţia *seekg* poziţionează pointerul de intrare (*get*) al fişierului, în timp ce funcţia *seekp* poziţionează pointerul de ieşire (*put*) al fişierului. Ambele funcţii specifică locaţia de poziţionat folosind un deplasament faţă de începutul, sfârşitul sau poziţia curentă a fişierului. După poziţionarea pointerului de fişier, operaţiile I/O pe fişiere se realizează prin funcţiile membru *read* sau *write*.

REALIZAREA IEŞIRII LA IMPRIMANTĂ

Există situaţii când ieşirea trebuie efectuată nu pe fişiere, ci la imprimantă. În general, imprimanta poate fi tratată ca un fişier de ieşire care va fi deschis cu numele de dispozitiv PRN.

```
ofstream  printer("PRN");
```

De exemplu, programul PRTFILE.CPP tipăreşte la imprimantă conţinutul fişierului specificat de primul argument al liniei de comandă:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   char text[256];

   ofstream printer("PRN");

   ifstream file(argv[1]);

   if (file.fail())
     cerr << "Error opening the file " << argv[1] << endl;
   else
     {
       while (! file.eof())
         {
           file.getline(text, sizeof(text));

           if (file.good())
```

```
        {
          printer << text << endl;

          if (printer.fail())
            {
              cerr << "Error writing to printer" << endl;
              exit(1);
            }
        }
      else if (! file.eof())
        {
          cerr << "Error reading from file " << argv[1] << endl;
          exit(1);
        }
    }
  file.close();
  printer.close();
 }
}
```

REZUMAT

Pe măsură ce programele devin mai complexe, ele vor scrie în şi vor citi informaţii din fişiere. După cum s-a văzut, C++ tratează fişierele foarte asemănător stream-urilor I/O prezentate în Capitolul 1. Înainte de a continua cu Capitolul 4, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Când deschideţi un fişier, de fapt specificaţi dacă realizaţi operaţii de intrare sau ieşire.
- ✓ Fişierul antet FSTREAM.H defineşte tipurile de clase *ifstream, ofstream* şi *fstream*. Clasa *ifstream* corespunde fişierelor de intrare. Clasa *ofstream* este folosită pentru fişiere de ieşire. În sfârşit, clasa *fstream* este folosită pentru fişiere pe care le deschideţi pentru ambele operaţii.
- ✓ Pentru a deschide un fişier, se poate folosi funcţia membru *open* sau funcţia constructor a clasei.
- ✓ După deschiderea unui fişier, pentru a realiza operaţii de intrare/ieşire se pot folosi atât operatorii de inserţie şi extracţie, cât şi funcţiile membru.
- ✓ După ce un fişier a fost folosit, el trebuie închis cu funcţia membru *close*.
- ✓ Prin folosirea manipulatorilor de stream-uri de fişier ca *setw, setprecision, setiosflags*, programele pot realiza ieşirea formatată pe fişiere. De asemenea, se pot scrie pe fişiere caractere speciale ca \a, \t, sau chiar \n.

- ✓ La deschiderea unui fişier pentru intrare sau ieşire, programele pot folosi specificatorii modului de deschidere pentru a controla modul în care se deschid fişierele. De exemplu, folosind specificatorul *ios::app*, se poate deschide un fişier existent pentru a fi extins.
- ✓ La efectuarea operaţiilor I/O pe fişiere sau la extinderea fişierelor, programele pot folosi funcţii membru, ca *fail* sau *good*, pentru a testa succesul fiecărei operaţii. În cazul unei nereuşite, programele o pot detecta şi pot reacţiona la eroare.
- ✓ Dacă programele scriu sau citesc din fişiere date de tip structuri, tablouri sau chiar date de tip real, ele trebuie să realizeze operaţii pe fişiere binare. Fişierele binare se deschid folosind modul *ios::binary*, iar operaţiile pe acestea se efectuează cu funcţiile membru *read* şi *write*.
- ✓ Operaţiile secvenţiale pe fişiere se desfăşoară de la începutul spre sfârşitul fişierului. Pe de altă parte, operaţiile cu acces aleator pe fişiere se pot efectua pe orice locaţie a unui fişier. Poziţionarea pointerilor de fişier pentru operaţii de citire aleatoare se face cu funcţia membru *seekg*. Analog, poziţionarea pointerilor de fişier pentru operaţii de scriere aleatoare se realizează cu funcţia *seekp*.

CAPITOLUL 4
ACOMODAREA CU MOŞTENIREA CLASELOR

În Capitolul 2 aţi învăţat că obiectele, care modelează realitatea înconjurătoare, pot avea caracteristici comune. Când definiţi clasele într-un program, veţi vedea că şi ele pot avea aceleaşi proprietăţi. Folosind acest avantaj al relaţiilor dintre obiecte, se poate reduce substanţial cantitatea de cod care trebuie scrisă. Reducerea codului înseamnă nu numai reducerea timpului de programare, dar şi îmbunătăţirea lizibilităţii programului; o cantitate mai mică de cod conduce la erori mai puţine.

Când construiţi clase bazate pe relaţii între obiecte, folosiţi conceptul de *moştenire* - o clasă derivată poate *moşteni* proprietăţile clasei de bază. În mod normal, proiectarea începe cu o clasă simplă, din care se vor construi clase mai complexe. De exemplu, s-ar putea începe cu o clasă *engine* din care se poate construi o clasă *boat engine*, o clasă *car engine* sau *motorcycle engine*. Acest capitol examinează moştenirea claselor C++ în detaliu, iar în momentul încheierii sale veţi cunoaşte:

- Ce este şi la ce foloseşte moştenirea
- Cum se folosesc funcţiile constructor în cazul moştenirii
- Care este „ştiinţa“ şi „arta“ programării pe obiecte
- Ce este moştenirea multiplă
- Ce este moştenirea pe mai multe niveluri
- Ce este membrul protejat al unei clase şi la ce foloseşte

CONCEPTUL DE MOŞTENIRE

Când definiţi clase într-un program, apar situaţii când două sau mai multe clase au caracteristici comune (aceiaşi membri). În loc de a reproduce membrii în fiecare clasă (consumând timp şi introducând o cantitate mare de cod care ar putea avea o mulţime de erori şi care necesită foarte mult timp pentru a-l depana), s-ar putea defini o clasă de bază care să conţină funcţiile membru comune. Apoi, s-ar putea deriva (construi) clase succesive din clasa de bază. Într-o astfel de abordare, clasele derivate vor moşteni caracteristicile clasei de bază.

ANALIZA UNUI EXEMPLU SIMPLU

Să presupunem, de exemplu, că scrieţi un program pentru un medic veterinar care doreşte să ţină înregistrările unor rase diferite de câini pe care îi tratează, precum şi bolile de care suferă aceştia. În mod concret, veterinarul doreşte să vadă ce tipuri de boli afectează câinii cu pete şi pe cei fără pete. Un mod de abordare poate fi crearea a două clase de câini, o clasă pentru câinii cu pete şi alta pentru cei fără pete, cum se arată mai jos:

```
class spotted_dogs {
  public:
    spotted_dogs(char *breed, int height, int weight, char *color,
       char *spot_color);
    void show_breed(void);
    void spot_info(void);
  private:
    char breed[64];
    int height;
    int weight;
    char color[64];
    char spot_color[64];
};

class unspotted_dogs {
  public:
    unspotted_dogs(char *breed, int height, int weight,
       char *color);
    void show_breed(void);
  private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};
```

Următorul program, TWODOGS.CPP, foloseşte aceste două clase pentru a crea obiectele de tip *spotted_dogs* şi *unspotted_dogs*:

```
#include <iostream.h>
#include <string.h>

class spotted_dogs {
  public:
    spotted_dogs(char *breed, int height, int weight, char *color,
       char *spot_color);
    void show_breed(void);
    void spot_info(void);
  private:
    char breed[64];
```



```
    int height;
    int weight;
    char color[64];
    char spot_color[64];
};

class unspotted_dogs {
  public:
    unspotted_dogs(char *breed, int height, int weight,
       char *color);
    void show_breed(void);
  private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};

spotted_dogs::spotted_dogs(char *breed, int height, int weight,
   char *color, char *spot_color)
 {
   strcpy(spotted_dogs::breed, breed);
   spotted_dogs::height = height;
   spotted_dogs::weight = weight;
   strcpy(spotted_dogs::color, color);
   strcpy(spotted_dogs::spot_color, spot_color);
 }

void spotted_dogs::show_breed(void)
 {
   cout << "Breed: " << breed << endl;
   cout << "Height: " << height << " Weight: " << weight << endl;
   cout << "Color: " << color << endl;
   cout << "Spot color: " << spot_color << endl << endl;
 }

void spotted_dogs::spot_info(void)
 {
   cout << breed << " has " << spot_color << " spots" << endl <<
           endl;
 }

unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
   char *color)
 {
   strcpy(unspotted_dogs::breed, breed);
   unspotted_dogs::height = height;
   unspotted_dogs::weight = weight;
   strcpy(unspotted_dogs::color, color);
```

```
}

void unspotted_dogs::show_breed(void)
{
  cout << "Breed: " << breed << endl;
  cout << "Height: " << height << " Weight: " << weight << endl;
  cout << "Color: " << color << endl << endl;
}

void main(void)
{
  spotted_dogs happy("Dalmatian", 24, 60, "white",
    "black or brown (liver)");
  unspotted_dogs rover("Labrador Retriever", 24, 65,
    "black or yellow");

  happy.show_breed();
  happy.spot_info();

  rover.show_breed();
}
```

După cum se poate vedea, programul atribuie valori variabilelor membru, apoi afişează aceste valori folosind funcţia *show_breed*. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> TWO_DOGS <ENTER>
Breed: Dalmatian
Height: 24 Weight: 60
Color: white
Spot color: black or brown (liver)

Dalmatian has black or brown (liver) spots

Breed: Labrador Retriever
Height: 24 Weight: 65
Color: black or yellow
```

Dacă priviţi cu atenţie cele două clase, veţi observa că ele conţin câţiva membri comuni. Folosind conceptul de moştenire, se poate crea o clasă ce conţine membrii comuni, apoi alte două clase mai mici ce conţin membrii specifici pentru câinii cu pete şi fără pete. Să creăm, pentru început, clasa *dogs* ce conţine membrii comuni:

```
class dogs {
 public:
   dogs(char *breed, int height, int weight, char *color);
   void show_breed(void);
 private:
```



```
   char breed[64];
   int height;
   int weight;
   char color[64];
};
```

Să creăm acum două clase mai mici bazate pe clasa *dogs*. Cu alte cuvinte, două clase care *moştenesc* caracteristicile (membrii) clasei *dogs*. Următoarele instrucţiuni, de exemplu, creează clasele folosind moştenirea:

```
class spotted_dogs : public dogs {
 public:
   spotted_dogs(char *breed, int height, int weight, char *color,
     char *spot_color);
   void show_breed(void);
   void spot_info(void);
 private:
   char spot_color[64];
};

class unspotted_dogs : public dogs {
 public:
   unspotted_dogs(char *breed, int height, int weight,
     char *color);
};
```

Observaţi semnul ":" situat după numele clasei şi urmat de cuvintele ***public dogs***. Semnul ":" informează compilatorul C++ că clasa definită moşteneşte caracteristicile clasei care urmează:

Clasă derivată

```
class spotted_dogs : public dogs {
```

Clasă de bază

După cum se observă, clasa *spotted_dogs* adaugă noi funcţii membru şi o variabilă membru. Clasa *unspotted_dogs* nu adaugă nici un membru nou, deoarece câinii fără pete nu au nevoie de alte caracteristici, în afara celor furnizate de clasa de bază *dogs*. Programul ar fi putut folosi, pur şi simplu, clasa *dogs*, la fiecare creare a unui obiect de tipul câine fără pete. Totuşi, pentru îmbunătăţirea clarităţii programului (pentru a permite programului să lucreze cu câini cu pete şi câini fără pete, în loc de câini cu pete şi câini), s-a derivat clasa *unspotted_dogs*.



SĂ ÎNŢELEGEM MOŞTENIREA CLASELOR

Când se definesc clase într-un program, apar situaţii când două sau mai multe clase au caracteristici comune (aceiaşi membri). În loc de a reproduce membrii în fiecare clasă, se poate defini o *clasă de bază* ce conţine funcţiile membru comune. Apoi, se pot construi (deriva) clase succesive din clasa de bază. În acest caz, clasele *derivate* moştenesc caracteristicile clasei de bază. Să presupunem, de exemplu, că definiţi următoarea clasă *vehicle*:

```
class vehicle {
   public:
      vehicle(char *name, int wheels, int engine);
      void show_vehicle(void);
   private:
      char name[64];

      int wheels;
      int engine;
 };
```

Folosind moştenirea, se poate declara o clasă *motorcycle* astfel:

```
class motorcycle : public vehicle {
   public:
      motorcycle(char *name, int wheels, int engine, int seats);
      void show_cycle(void);
   private:
      int seats;
 };
```

În acelaşi mod, se poate declara şi clasa *automobile*:

```
class automobile: public vehicle {
   public:
      automobile(char *name, int wheels, int engine, int doors);
      void show_auto(void);
   private:
      int doors;
 };
```

În acest caz, clasele *motorcycle* şi *automobile* moştenesc amândouă caracteristicile definite în clasa *vehicle*.

FOLOSIREA FUNCŢIILOR CONSTRUCTOR LA MOŞTENIREA CLASELOR

Aşa cum aţi învăţat în Capitolul 2, funcţiile constructor permit programelor să iniţializeze variabilele membru ale claselor. Tot constructorii se folosesc pentru iniţializări şi în cazul moştenirii claselor. Singura diferenţă este aceea că, constructorul clasei derivate trebuie să apeleze constructorul clasei de bază. Următoarea funcţie implementează funcţia constructor a clasei de bază *dogs*:

```
dogs::dogs(char *breed, int height, int weight, char *color)
 {
   strcpy(dogs::breed, breed);
   dogs::height = height;
   dogs::weight = weight;
   strcpy(dogs::color, color);
 }
```

Funcţiile constructor ale claselor derivate folosesc o sintaxă similară definiţiei clasei, având simbolul ":" urmat de funcţia constructor a clasei de bază:

Constructor clasă derivată

```
spotted_dogs::spotted_dogs(char *breed, int height, int weight,
   char *color, char *spot_color) : dogs (breed, height, weight,
   color)
 {
   strcpy(spotted_dogs::spot_color, spot_color);
 }
```

Constructor clasă de bază

```
unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
   char *color) : dogs(breed, height, weight, color)
 {
   // Do nothing-base class constructor initialized members
 }
```

Aşa cum se poate vedea, apelul constructorului clasei de bază foloseşte nume de parametri identici celor transferaţi constructorului clasei derivate.

Parametrii clasei derivate

```
spotted_dogs::spotted_dogs(char *breed, int height, int weight,
   char *color, char *spot_color) : dogs (breed, height, weight,
   color)
 {
   strcpy(spotted_dogs::spot_color, spot_color);
 }
```

Parametrii clasei de bază

```
unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
   char *color) : dogs(breed, height, weight, color)
 {
   // Do nothing-base class constructor initialized members
 }
```

Când programul apelează constructorul clasei *spotted_dogs*, C++ va apela automat, mai întâi, constructorul clasei *dogs*. Funcţia constructor a clasei de bază este întotdeauna apelată înaintea constructorului clasei derivate. În acest mod, membrii clasei de bază sunt corect iniţializaţi, iar memoria va fi corect alocată pentru buffer-ele clasei de bază, înainte de referirea clasei derivate.

Dacă examinaţi funcţia constructor a clasei *spotted_dogs*, veţi vedea că iniţializează variabila membru *spot_color*. În schimb, constructorul *unspotted_dogs* nu are valori de atribuit (în afara celor folosite de constructorul clasei de bază).

Următorul program, NEW_DOGS.CPP foloseşte moştenirea pentru a crea clasele de câini cu pete şi fără pete.

```
#include <iostream.h>
#include <string.h>

class dogs {
  public:
    dogs(char *breed, int height, int weight, char *color);
    void show_breed(void);
  private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};

class spotted_dogs : public dogs {
  public:
    spotted_dogs(char *breed, int height, int weight, char *color,
       char *spot_color);
    void show_breed(void);
    void spot_info(void);
  private:
    char spot_color[64];
};

class unspotted_dogs : public dogs {
  public:
    unspotted_dogs(char *breed, int height, int weight,
       char *color);
};

dogs::dogs(char *breed, int height, int weight, char *color)
 {
   strcpy(dogs::breed, breed);
   dogs::height = height;
   dogs::weight = weight;
   strcpy(dogs::color, color);
 }
```



```
spotted_dogs::spotted_dogs(char *breed, int height, int weight,
  char *color, char *spot_color) : dogs (breed, height, weight,
  color)
 {
   strcpy(spotted_dogs::spot_color, spot_color);
 }

unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
  char *color) : dogs(breed, height, weight, color)
 {
   // Do nothing-base class constructor initialized members
 }

void dogs::show_breed(void)
 {
   cout << "Breed: " << breed << endl;
   cout << "Height: " << height << " Weight: " << weight << endl;
   cout << "Color: " << color << endl;
 }

void spotted_dogs::show_breed(void)
 {
   dogs::show_breed();
   cout << "Spot color: " << spot_color << endl << endl;
 }

void spotted_dogs::spot_info(void)
 {
   cout << "This breed has " << spot_color << " spots" << endl <<
         endl;
 }

void main(void)
 {
   spotted_dogs happy("Dalmatian", 24, 60, "white",
      "black or brown (liver)");
   unspotted_dogs rover("Labrador Retriever", 24, 65,
      "black or yellow");

   happy.show_breed();
   happy.spot_info();

   rover.show_breed();
 }
```

Deoarece clasa *spotted_dogs* conţine membrul *spot_color*, clasa foloseşte funcţia membru *show_breed* proprie. Folosind operatorul de rezoluţie globală (::) funcţia apelează mai întâi funcţia *show_breed* a clasei de bază pentru a afişa variabilele membru comune tuturor raselor de câini. Apoi, funcţia afişează valoarea propriei variabile membru:

```
void spotted_dogs::show_breed(void)
{
   dogs::show_breed();
   cout << "Spot color: " << spot_color << endl << endl;
}
```

Apelarea funcţiei clasei de bază

Afişarea variabilei membru **spot_color**

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> NEW_DOGS  <ENTER>
Breed: Dalmatian
Height: 24 Weight: 60
Color: white
Spot color: black or brown (liver)
This breed has black or brown (liver) spots

Breed: Labrador Retriever
Height: 24 Weight: 65
Color: black or yellow
```

Să observăm că în ieşirea programului s-a schimbat următoarea linie:

```
Dalmatian has black or brown (liver) spots

This breed has black or brown (liver) spots
```

Programul foloseşte funcţia *spot_info* pentru a afişa mesajul corespunzător:

```
void spotted_dogs::spot_info(void)
{
   cout << "This breed has " << spot_color << " spots" << endl <<
         endl;
}
```

În acest caz, funcţia nu poate folosi variabila *breed* a clasei *dogs*, care conţine numele rasei (Dalmatian), deoarece aceasta nu este publică în clasa *dogs*. Singurul mod prin care o clasă derivată poate avea acces la membrii privaţi ai clasei de bază este de a folosi funcţiile membru ale clasei de bază, cum este *show_breed*. Un alt mod de a rezolva această problemă este de a face variabila membru *breed* publică în clasa *dogs*. De asemenea, aşa cum veţi vedea mai târziu, se poate declara variabila ca fiind un membru *protejat (protected)* al clasei, ceea ce va permite accesul direct numai prin intermediul obiectelor clasei derivate sau ale clasei de bază.

ANALIZA UNUI EXEMPLU

Pentru a înţelege mai bine cum şi când sunt apelate funcţiile constructor şi destructor în situaţia moştenirii claselor, următorul program, GENERIC.CPP, defineşte două clase, denumite *base* şi *derived*. În cadrul fiecărui constructor sau destructor, programul afişează un mesaj ce identifică funcţia curentă:

```
#include <iostream.h>
#include <string.h>
```



```
class base {
 public:
   base(char *base_message);
   ~base(void);
 private:
   char base_message[64];
};

class derived : public base {
 public:
   derived(char *derived_message);
   ~derived(void);
 private:
   char derived_message[64];
};

base::base(char *message)
 {
   strcpy(base_message, message);
   cout << "In base class constructor: " << message << endl;
 }

base::~base(void)
 {
   cout << "In base class destructor: " << base_message << endl;
 }

derived::derived(char *message) : base("Hello, base")
 {
   strcpy(derived_message, message);
   cout << "In derived class constructor: " << message << endl;
 }

derived::~derived(void)
 {
   cout << "In derived class destructor: " << derived_message <<
         endl;
 }

void main(void)
 {
   derived object("Hello, world");
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> GENERIC  <ENTER>
In base class constructor: Hello, base
In derived class constructor: Hello, world
In derived class destructor: Hello, world
In base class destructor: Hello, base
```

După cum se vede, când programul instanţiază obiectul derivat, este apelată mai întâi funcţia clasei de bază, urmată de constructorul clasei derivate. La distrugerea obiectului, secvenţa de apel se desfăşoară invers, fiind apelat mai întâi destructorul clasei derivate, apoi destructorul clasei de bază.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM CUVÂNTUL – CHEIE PUBLIC

Când se creează o clasă derivată, se specifică numele acesteia, urmat de simbolul ":" şi de numele clasei de bază, ca mai jos:

```
class derived : public base {
   // Membrii
};
```

După cum se observă, definiţia clasei derivate foloseşte cuvântul-cheie *public*. Aceasta înseamnă că membrii de tip *public* ai clasei de bază sunt consideraţi de acelaşi tip şi în clasa derivată. În acelaşi mod, membrii protejaţi (*protected*) ai clasei de bază sunt trataţi ca membri protejaţi în clasa derivată. Majoritatea programelor vor folosi cuvântul-cheie *public* ca mai sus. Dacă la definiţia unei clase se foloseşte cuvântul-cheie *private*, atunci membrii publici şi protejaţi ai clasei de bază vor fi trataţi drept membri nonpublici (*private*) în clasa derivată.

ANALIZA ALTUI EXEMPLU

Când vă gândiţi la o unitate de disc, îi asociaţi întotdeauna anumite atribute, precum: capacitatea de memorare, geometria discului (numărul de feţe, piste, sectoare etc.), sau viteza de transfer. Aşa cum se arată în Figura 4.1, aceste caracteristici sunt moştenite atât de hard-disk-uri cât şi de dischete, fiecare dintre ele având şi caracteristici proprii, ca tipul de interfaţă (SCSI sau IDE) sau mecanismul de protecţie la scriere.

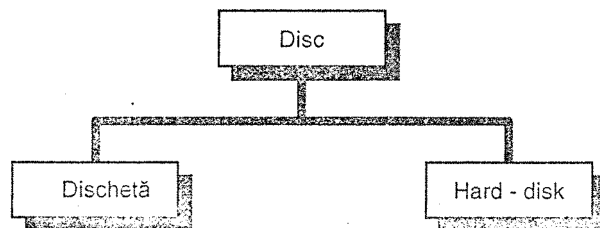


Figura 4.1. Relaţia dintre tipurile de disc

Următorul program, DISKCLAS.CPP, creează o clasă de bază numită *disk*:


```
class disk {
 public:
   disk(char *name, int sides, int tracks, int sectors_per_track,
     int bytes_per_sector);
   void show_disk(void);
 private:
   char name[64];
   int sides;
   int tracks;
   int sectors_per_track;
   int bytes_per_sector;
   long capacity;
};
```

Apoi, programul derivează clasele *hard_disk* şi *floppy_disk*, ca mai jos:

```
class floppy_disk : public disk {
 public:
   floppy_disk(char *name, int sides, int tracks,
     int sectors_per_track, int bytes_per_sector, int state);
   void set_write_protect(int state);
   void show_floppy(void);
 private:
   int write_protect_state;
};

class hard_disk : public disk {
 public:
   hard_disk(char *name, int sides, int tracks,
     int sectors_per_track, int bytes_per_sector,
     char *controller_type);
   void show_hard_disk(void);
 private:
   char controller_type[64];
};
```

Programul DISKCLAS.CPP este redat în întregime în continuare:

```
#include <iostream.h>
#include <string.h>

class disk {
 public:
   disk(char *name, int sides, int tracks, int sectors_per_track,
     int bytes_per_sector);
   void show_disk(void);
 private:
   char name[64];
   int sides;
   int tracks;
   int sectors_per_track;
```

```
   int bytes_per_sector;
   long capacity;
};

class floppy_disk : public disk {
  public:
    floppy_disk(char *name, int sides, int tracks,
      int sectors_per_track, int bytes_per_sector, int state);
    void set_write_protect(int state);
    void show_floppy(void);
  private:
    int write_protect_state;
};

class hard_disk : public disk {
  public:
    hard_disk(char *name, int sides, int tracks,
      int sectors_per_track, int bytes_per_sector,
      char *controller_type);
    void show_hard_disk(void);
  private:
    char controller_type[64];
};

disk::disk(char *name, int sides, int tracks, int sectors_per_track,
   int bytes_per_sector)
 {
   strcpy(disk::name, name);
   disk::sides = sides;
   disk::tracks = tracks;
   disk::sectors_per_track = sectors_per_track;
   disk::bytes_per_sector = bytes_per_sector;
   capacity = (long) sides * (long) tracks;
   capacity *= (long) sectors_per_track * (long) bytes_per_sector;
 }

void disk::show_disk(void)
 {
   cout << "Disk name: " << name << endl;
   cout << "Sides: " << sides << " Tracks: " << tracks << endl;
   cout << "Sectors per track: " << sectors_per_track << endl;
   cout << "Bytes per sector: " << bytes_per_sector << endl;
   cout << "Capacity: " << capacity << " bytes" << endl;
 }

floppy_disk::floppy_disk(char *name, int sides, int tracks,
   int sectors_per_track, int bytes_per_sector, int state) :
   disk (name, sides, tracks, sectors_per_track, bytes_per_sector)
```



```
 {
   write_protect_state = state;
 }

void floppy_disk::show_floppy(void)
 {
   show_disk();
   cout << "Write-protect is: " << ((write_protect_state) ?
           "On": "Off") << endl << endl;
 }

hard_disk::hard_disk(char *name, int sides, int tracks,
   int sectors_per_track, int bytes_per_sector, char *controller) :
   disk(name, sides, tracks, sectors_per_track, bytes_per_sector)
 {
   strcpy(controller_type, controller);
 }

void hard_disk::show_hard_disk(void)
 {
   show_disk();
   cout << "Controller type: " << controller_type << endl << endl;
 }

void main(void)
 {
   hard_disk ide_drive("Hard disk", 4, 615, 80, 512, "IDE");
   floppy_disk high_density("Floppy disk", 2, 80, 18, 512, 1);

   ide_drive.show_hard_disk();
   high_density.show_floppy();
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> DISKCLAS  <ENTER>
Disk name: Hard disk
Sides: 4 Tracks: 615
Sectors per track: 80
Bytes per sector: 512
Capacity: 100761600 bytes
Controller type: IDE

Disk name: Floppy disk
Sides: 2 Tracks: 80
Sectors per track: 18
Bytes per sector: 512
Capacity: 1474560 bytes
Write-protect is: On
```

UN ULTIM EXEMPLU DE MOŞTENIRE PE UN SINGUR NIVEL

Probabil că deja v-aţi obişnuit cu sintaxa claselor de bază şi a celor derivate. Sintaxa reprezintă "ştiinţa" programării orientate pe obiecte. "Arta" acestei programări începe când trebuie să determinăm care din atribute aparţin claselor de bază, şi care atribute trebuie să figureze în clasele derivate. Cea mai bună metodă de a stăpâni "arta" proiectării orientate pe obiect este de a exersa în permanenţă, de a rafina treptat conceptele introduse. Să analizăm din nou un exemplu. Să presupunem că trebuie să creaţi o bază de date a angajaţilor unei mari companii. Compania are manageri cu salariu anual, angajaţi permanenţi plătiţi cu ora, şi angajaţi temporari plătiţi cu ora. Pentru fiecare tip de angajat, programul trebuie să urmărească următoarele informaţii:

Salariaţii	**Permanent cu ora**	**Temporar cu ora**
nume	nume	nume
telefon acasă	telefon acasă	telefon acasă
telefon serviciu	telefon serviciu	telefon serviciu
salariu anual	plata pe oră	plata pe oră
bonificaţii		
superior ierarhic	superior ierarhic	superior ierarhic
asistent		

Prima "mişcare" care se face la proiectarea clasei de bază este de a selecta atributele comune, ca mai jos:

Salariaţii	**Permanenţii cu ora**	**Temporarii cu ora**
nume	nume	nume
telefon acasă	telefon acasă	telefon acasă
telefon serviciu	telefon serviciu	telefon serviciu
salariu anual	plata pe oră	plata pe oră
bonificaţii		
superior ierarhic	superior ierarhic	superior ierarhic
asistent		

Folosind atributele comune, se poate acum crea clasa de bază *employee*, cum se arată mai jos:

```
class employee {
  public:
    employee(char *name, char *home_phone, char *office_phone,
```



```
      char *reports_to);
    void show_employee(void);
  private:
    char name[64];
    char home_phone[64];
    char office_phone[64];
    char reports_to[64];
};
```

Apoi, înlăturând atributele comune, clasele derivate devin:

```
class salaried : public employee {
  public:
    salaried(char *name, char *home_phone, char *office_phone,
      char *reports_to, float salary, float bonus_level,
      char *assistant);
    void show_salaried(void);
  private:
    float salary;
    float bonus_level;
    char assistant[64];
};

class hourly : public employee {
  public:
    hourly(char *name, char *home_phone, char *office_phone,
      char *reports_to, float wage);
    void show_hourly(void);
  private:
    float wage;
};

class temporary : public employee {
  public:
    temporary(char *name, char *home_phone, char *office_phone,
      char *reports_to, float wage);
    void show_temporary(void);
  private:
    float wage;
};
```

Dacă examinaţi clasele *hourly* şi *temporary*, veţi vedea că membrii lor sunt identici. Tocmai aici arta, preferinţele programatorului şi alţi asemenea factori intră în rol.

În funcţie de modul în care programul lucrează cu obiectele, se poate crea o variabilă membru în clasa *hourly* cu rol de indicator, pentru a preciza dacă un obiect corespunde unui angajat permanent sau temporar, ca mai jos:

```
enum worker_type { permanent, temporary };

class hourly : public employee {
```

```
  public:
    temporary(char *name, char *home_phone, char *office_phone,
       char *reports_to, float wage, worker_type flag);
    void show_temporary(void);
  private:
    float wage;
    worker_type flag;
};
```

În cadrul funcţiilor membru ale clasei, codul acestora trebuie să examineze variabila indicator pentru a determina tipul corect.

Există unele situaţii în care se doreşte restrângerea anumitor operaţii din program la angajaţii temporari sau permanenţi. În astfel de situaţii obiectele tip angajat nu mai sunt aceleaşi, iar programul trebuie să folosească două tipuri distincte de clase. În acest fel, claritatea programului se va îmbunătăţi. De exemplu, următoarele fragmente ilustrează o funcţie care limitează operaţiile numai asupra angajaţilor temporari cu ora:

```
float temporary::pay_overtime(void)
  {
      // Statements
  }
```

```
float hourly::pay_overtime(void)
  {
     if (flag == temporary)
        {
           // Statements
        }
  }
```

Prin simpla examinare a antetului f...ţiei *pay-overtime* din partea stângă, o persoană care citeşte programul poate imediat să determine că funcţia se aplică numai obiectelor de tip angajat temporar. În cazul funcţiei din partea dreaptă, cititorul programului trebuie să aibă acces la instrucţiunile sursă (în particular, instrucţiunea *if*) pentru a afla că funcţia afectează numai obiectele de tip permanent.

Următorul program, WORKERS.CPP, foloseşte clasele *employee, salaried, permanent* şi *temporary*:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    employee(char *name, char *home_phone, char *office_phone,
       char *reports_to);
    void show_employee(void);
  private:
    char name[64];
    char home_phone[64];
    char office_phone[64];
```



```
    char reports_to[64];
};

class salaried : public employee {
  public:
    salaried(char *name, char *home_phone, char *office_phone,
       char *reports_to, float salary, float bonus_level,
       char *assistant);
    void show_salaried(void);
  private:
    float salary;
    float bonus_level;
    char assistant[64];
};

class hourly : public employee {
  public:
    hourly(char *name, char *home_phone, char *office_phone,
       char *reports_to, float wage);
    void show_hourly(void);
  private:
    float wage;
};

class temporary : public employee {
  public:
    temporary(char *name, char *home_phone, char *office_phone,
       char *reports_to, float wage);
    void show_temporary(void);
  private:
    float wage;
};

employee::employee(char *name, char *home_phone, char *office_phone,
   char *reports_to)
 {
   strcpy(employee::name, name);
   strcpy(employee::home_phone, home_phone);
   strcpy(employee::office_phone, office_phone);
   strcpy(employee::reports_to, reports_to);
 }

void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Home phone: " << home_phone << endl;
   cout << "Office phone: " << office_phone << endl;
   cout << "Reports to: " << reports_to << endl;
 }
```

```
salaried::salaried(char *name, char *home_phone, char *office_phone,
   char *reports_to, float salary, float bonus_level,
   char *assistant) : employee(name, home_phone, office_phone,
   reports_to)
 {
   salaried::salary = salary;
   salaried::bonus_level = bonus_level;
   strcpy(salaried::assistant, assistant);
 }

void salaried::show_salaried(void)
 {
    show_employee();
    cout << "Salary: $" << salary << endl;
    cout << "Bonus level: $" << bonus_level << endl;
    cout << "Assistant: " << assistant << endl;
 }

hourly::hourly(char *name, char *home_phone, char *office_phone,
   char *reports_to, float wage) : employee(name, home_phone,
   office_phone, reports_to)
 {
    hourly::wage = wage;
 }

void hourly::show_hourly(void)
 {
    show_employee();
    cout << "Wage: $" << wage << endl;
 }

temporary::temporary(char *name, char *home_phone,
   char *office_phone, char *reports_to, float wage) :
   employee(name, home_phone, office_phone, reports_to)
 {
    temporary::wage = wage;
 }

void temporary::show_temporary(void)
 {
    show_employee();
    cout << "Wage: $" << wage << endl;
 }

void main(void)
 {
   salaried top_boss("Joe Smith", "555-1111", "555-1112",
     "Mark Jones", 30000.0, 10000.0, "Alicia Jones");

   hourly typist("David Kline", "555-2222", "555-2223",
      "John Martin", 4.50);
```



```
   temporary receptionist("Mary Scott", "555-3333", "555-3334",
      "Doris Davis", 4.00);

   top_boss.show_salaried();

   cout << endl << endl;

   typist.show_hourly();

   cout << endl << endl;

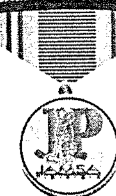
   receptionist.show_temporary();
 }
```

La compilarea și execuția acestui program, pe ecran vor apărea următoarele rezultate:

```
C:\> WORKERS  <ENTER>
Name: Joe Smith
Home phone: 555-1111
Office phone: 555-1112
Reports to: Mark Jones
Salary: $30000
Bonus level: $10000
Assistant: Alicia Jones

Name: David Kline
Home phone: 555-2222
Office phone: 555-2223
Reports to: John Martin
Wage: $4.5

Name: Mary Scott
Home phone: 555-3333
Office phone: 555-3334
Reports to: Doris Davis
Wage: $4
```



PROTOTIPURI, REVIZII, FINALIZĂRI

Programarea orientată pe obiecte este atât artă, cât și știință. După ce ați scris primele programe, știința programării orientate pe obiecte devine destul de clară. Arta proiectării programelor constă în atribuirea corectă a membrilor către clasele de bază sau derivate.

De regulă, se consideră prima clasă proiectată și membrii ei drept un prototip. Pe măsură ce dezvoltați programul, revedeți și actualizați membrii claselor. În majoritatea cazurilor, aceasta duce la eliminarea rescrierii codului într-o etapă ulterioară, cauzată de o proiectare inițială incorectă.

În general, dacă vă găsiţi în situaţia de a transmite parametri unor funcţii ce conţin informaţii suplimentare despre un obiect, atunci trebuie să refaceţi toată proiectarea. Dacă în schimb extindeţi codul, atunci rămâne o uşă deschisă pentru erori, iar claritatea programului va avea de suferit.

SĂ ÎNŢELEGEM MOŞTENIREA MULTIPLĂ

Există situaţii în care unele clase folosesc caracteristicile a două sau a mai multor clase de bază. Moştenirea multiplă este folosirea a două sau mai multor clase de bază pentru derivarea altei clase. Să presupunem, de exemplu, că s-au declarat clasele *book* şi *disk*:

```
class book {
  public:
    book(char *title, char *author, int pages);
    void show_book(void);
  private:
    char title[64];
    char author[64];
    int pages;
};

class disk {
  public:
    disk(float capacity);
    void show_disk(void);
  private:
    float capacity;
};

class bundle : public book, public disk {
  public:
    bundle(char *title, char *author, int pages, float capacity,
       float price);
    void show_bundle(void);
  private:
    float price;
};
```

Se poate apoi crea o clasă denumită *bundle* care este o combinaţie între cele două clase anterioare:

```
class bundle : public book, public disk {
  public:
    bundle(char *title, char *author, int pages, float capacity,
       float price);
    void show_bundle(void);
```



```
  private:
    float price;
};
```

În acest caz, *bundle* este clasa derivată, iar *disk* şi *book* sunt clasele de bază:

Clasă derivată

```
class bundle : public book, public disk {
```

Clase de bază

Când o clasă este derivată din două sau mai multe clase, clasele de la bază se separă prin virgule, cum s-a arătat mai sus. La declararea funcţiei constructor al clasei, trebuie specificat constructorul clasei de bază într-un mod similar:

Constructor clasă derivată

```
bundle::bundle(char *title, char *author, int pages, float capacity,
       float price) : book(title, author, pages) , disk(capacity)
 {
   bundle::price = price;
 }
```

Constructori clasă de bază

În acest caz, C++ va apela mai întâi constructorul *book*, urmat de funcţia *disk* şi în final de funcţia *bundle*.

Următorul program, MULT_INH.CPP, ilustrează moştenirea multiplă prin crearea clasei *bundle*. Programul nu face altceva decât să apeleze funcţiile constructor şi destructor, care la rândul lor afişează mesaje pentru urmărirea execuţiei:

```
#include <iostream.h>
#include <string.h>

class book {
  public:
    book(char *title, char *author, int pages);
    void show_book(void);
  private:
    char title[64];
    char author[64];
    int pages;
};

class disk {
  public:
    disk(float capacity);
    void show_disk(void);
  private:
    float capacity;
};
class bundle : public book, public disk {
```

```
  public:
    bundle(char *title, char *author, int pages, float capacity,
       float price);
    void show_bundle(void);
  private:
    float price;
};

book::book(char *title, char *author, int pages)
 {
   strcpy(book::title, title);
   strcpy(book::author, author);
   book::pages = pages;
 }

void book::show_book(void)
 {
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Pages: " << pages << endl;
 }

disk::disk(float capacity)
 {
   disk::capacity = capacity;
 }

void disk::show_disk(void)
 {
   cout << "Capacity: " << capacity << "Mb" << endl;
 }

bundle::bundle(char *title, char *author, int pages, float capacity,
   float price) : book(title, author, pages) , disk(capacity)
 {
   bundle::price = price;
 }

void bundle::show_bundle(void)
 {
   show_book();
   show_disk();
   cout << "Price: $" << price << endl;
 }

void main(void)
 {
   bundle this_book("Jamsa's 1001 C/C++ Tips", "Jamsa", 896,
    1.44, 39.95);

    this_book.show_bundle();
 }
```


La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> MULT_INH  <ENTER>
Title: Jamsa's 1001 C/C++ Tips
Author: Jamsa
Pages: 896
Capacity: 1.44Mb
Price: $39.95
```

După cum se poate vedea, programul apelează funcţiile constructor ale claselor *book, disk* şi *bundle*, în această ordine.



SĂ ÎNŢELEGEM MOŞTENIREA MULTIPLĂ

Moştenirea multiplă constă în folosirea a două sau a mai multor clase de bază entru derivarea unei clase noi. La derivarea noii clase din clasele de bază, trebuie specificat numele clasei derivate, urmat de numele claselor de bază, cum se arată mai jos:

```
class derived : public base1, public base2;
{
   // members
}
```

Fiecare din clasele de bază este precedată de cuvântul-cheie *public*. Acesta permite ca în clasele derivate, membrii *public* şi *protected* ai claselor de bază să fie trataţi respectiv ca membri *public* şi *protected* ai clasei derivate. La definirea constructorului clasei derivate se specifică şi constructorul clasei de bază, ca în exemplu următor:

```
derived::derived(int a, int b): base1(a), base2(b)
{
   // Statements
}
```

În acest caz, când programul va apela constructorul clasei *derived*, C++ va apela mai întâi constructorul clasei *base1*, apoi constructorul clasei *base2*.

SĂ ÎNŢELEGEM MOŞTENIREA PE MAI MULTE NIVELE

Aşa cum am văzut, moşternirea multiplă constă în folosirea a două sau mai multor clase la bază pentru a deriva o nouă clasă. ***Moştenirea pe mai multe nivele***, pe de altă parte, are loc la derivarea unei clase dintr-o clasă de bază care, la rândul ei, este derivată din altă clasă. De exemplu, Figura 4.2 ilustrează clasa *manager*, bazată pe clasa *worker*, aceasta fiind la rândul ei bazată pe clasa *person*.

Pentru a crea cele 3 clase, începem cu clasa *person*, ca mai jos:

```
class person {
  public:
    person(char *name, int age);
    void show_person(void);
  private:
    char name[64];
    int age;
};
```

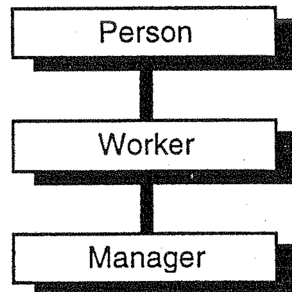


Figura 4.2. O moştenire pe nivele multiple.

Apoi, definim clasa *worker* astfel:

```
class worker : public person {
  public:
    worker(char *name, int age, char *phone, float wage);
    void show_worker(void);
  private:
    char phone[64];
    float wage;
};
```

În sfârşit, vom defini clasa *manager*:

```
class manager : public worker {
  public:
    manager(char *name, int age, char *phone, float wage,
       char *office);
    void show_manager(void);
  private:
    char office[64];
};
```

Următorul program, MULTILVL.CPP ilustrează moştenirea pe nivele multiple:

```
#include <iostream.h>
#include <string.h>

class person {
  public:
```



```
    person(char *name, int age);
    void show_person(void);
  private:
    char name[64];
    int age;
};

class worker : public person {
  public:
    worker(char *name, int age, char *phone, float wage);
    void show_worker(void);
  private:
    char phone[64];
    float wage;
};

class manager : public worker {
  public:
    manager(char *name, int age, char *phone, float wage,
       char *office);
    void show_manager(void);
  private:
    char office[64];
};

person::person(char *name, int age)
 {
   strcpy(person::name, name);
   person::age = age;
 }

void person::show_person(void)
 {
   cout << endl << "Name: " << name << endl;
   cout << "Age: " << age << endl;
 }
worker::worker(char *name, int age, char *phone, float wage) :
   person(name, age)
 {
   strcpy(worker::phone, phone);
   worker::wage = wage;
 }

void worker::show_worker(void)
 {
   show_person();
   cout << "Phone: " << phone << endl;
 }

manager::manager(char *name, int age, char *phone, float wage,
   char *office) : worker(name, age, phone, wage)
```

```
{
  strcpy(manager::office, office);
}

void manager::show_manager(void)
{
  show_worker();
  cout << "Office: " << office << endl;
}

void main(void)
{
  worker security("Ken Smith", 43, "555-1212", 4.50);

  manager boss("Betty Louis", 30, "555-2121", 12.50, "Room 3B");

  security.show_worker();

  boss.show_manager();
}
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> MULTILVL  <ENTER>
Name: Ken Smith
Age: 43
Phone: 555-1212

Name: Betty Louis
Age: 30
Phone: 555-2121
Office: Room 3B
```

SĂ ÎNŢELEGEM MEMBRII DE TIP PROTECTED AI UNEI CLASE

Aşa cum aţi învăţat în Capitolul 2, funcţiile membru ale clasei se pot declara de tip *public* sau *private*. La declararea unui membru ca *public*, acesta va fi accesibil în tot programul (când obiectul este în domeniu - vezi Capitolul 9). Un membru de tipul *private* poate fi accesibil numai prin funcţiile membru ale clasei. În programele din acest capitol, clasele derivate au avut acces numai la membrii de tip *public* ai clasei de bază. De exemplu, următorul program, INPUBLIC.CPP, ilustrează modul în care o clasă derivată poate avea acces la membrii de tip *public* ai clasei de bază. În acest caz, programul are acces la membrul de tip public *base_number* al clasei de bază, dar nu are acces la membrul nonpublic *base_message*:

```
#include <iostream.h>
#include <string.h>
```



```
class base {
  public:
    base(char *base_message, int number);
    int base_number;
  private:
    char base_message[64];
};

class derived : public base {
  public:
    derived(char *derived_message, int number);
  private:
    char derived_message[64];
};

base::base(char *message, int number)
 {
   strcpy(base_message, message);
   base_number = number;
 }

derived::derived(char *message, int number) : base("Base message",
   number)
 {
   strcpy(derived_message, message);
 }

void main(void)
 {
   derived object("Hello, world", 1001 );

   cout << "The base number is " << object.base_number << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> INPUBLIC  <ENTER>
The base number is 1001
```

Exersaţi cu acest program astfel încât funcţia *main* să facă referinţă la variabila membru *base_message*, în loc de *base_number*. Deoarece membrii nonpublici nu sunt accesibili în afara clasei, nici obiectele derivate nu vor avea acces la aceştia.

În funcţie de intenţia programului, pot apărea situaţii când se doreşte ca unii membri ai clasei de bază să poată fi referiţi de obiectele clasei derivate, fiind totuşi protejaţi de restul programului. În aceste cazuri se pot folosi în program membri protejaţi. Un membru protejat (*protected*) al unei clase poate fi direct folosit de clasele derivate, dar nu şi de alte părţi ale programului. Următorul program, PROTECT.CPP, ilustrează folosirea membrilor protejaţi ai unei clase:

```
#include <iostream.h>
#include <string.h>

class base {
  public:
    base(char *base_message, int number);
  protected:
    int base_number;
  private:
    char base_message[64];
};

class derived : public base {
  public:
    derived(char *derived_message, int number);
    void show_number(void);
  private:
    char derived_message[64];
};

base::base(char *message, int number)
 {
   strcpy(base_message, message);
   base_number = number;
 }

derived::derived(char *message, int number) : base("Base message",
   number)
 {
   strcpy(derived_message, message);
 }

void derived::show_number(void)
 {
   cout << "The base-class number is " << base_number << endl;
 }

void main(void)
 {
   derived object("Hello, world", 1001 );

   object.show_number();
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> PROTECT  <ENTER>
The base-class number is 1001
```

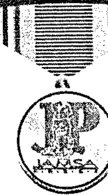
Exersaţi acest program. De exemplu, încercaţi să adăugaţi următoarele instrucţiuni la funcţia *main*:


```
void main(void)
 {
   derived object("Hello, world", 1001 );

   object.show_number();

   cout << "The base-class number is " << base_number << endl;
 }
```

Deoarece programul nu are acces la membrii protejaţi ai clasei, compilatorul va genera erori de sintaxă.



SĂ ÎNŢELEGEM MEMBRII PROTEJAŢI AI UNEI CLASE

Clasele derivate nu au acces la membrii de tip *private* ai clasei de bază. Există situaţii când clasele derivate trebuie să primească drepturi speciale de acces la membrii clasei de bază. Atunci se pot folosi membrii protejaţi ai clasei de bază, care sunt accesibili de către clasele derivate, dar nu şi din alte părţi ale programului. De exemplu, dată fiind următoarea definiţie de clasă, clasele derivate au acces la membrii *filename* şi *size*, dar nu şi la variabila membru *some_value* de tip *private*:

```
class base {
    public:
        base(char name[64], int value);
        void show_base(void);
   protected:
        char filename[64];
        long size;
   private:
        int some_value;
};
```

REZUMAT

Moştenirea constă în folosirea unei clase de bază pentru a deriva o nouă clasă. Moştenirea multiplă constă în folosirea a două sau mai multor clase pentru a deriva o altă clasă. Aşa cum aţi învăţat în acest capitol, C++ suportă complet atât moştenirea simplă, cât şi moştenirea multiplă. Folosind avantajul moştenirii, se poate economisi cod de program prin exploatarea relaţiilor între obiecte. Moştenirea este un concept cheie folosit în proiectarea şi programarea orientată pe obiecte. Înainte de a trece la Capitolul 5, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Moştenirea are loc atunci când o clasă derivată partajează (moşteneşte) caracteristici (membri) similare cu clasa de bază. Aceasta vă permite să economisiţi timp şi programe.
- ✓ La folosirea moştenirii, se pot utiliza în continuare constructori la iniţializarea claselor, dar constructorul clasei derivate trebuie să apeleze constructorul clasei de bază. Funcţiile constructor al clasei derivate folosesc o sintaxă similară definiţiei clasei, cu simbolul „:" urmat de constructorul clasei de bază.
- ✓ Sintaxa este „ştiinţa" programării orientată pe obiecte. „Arta" acestui tip de programare începe atunci când trebuie să determinaţi atributele ce aparţin clasei de bază şi cele care aparţin claselor derivate.
- ✓ La proiectarea claselor, pot apărea situaţii când anumite clase folosesc caracteristicile a două sau mai multor clase de bază. Aceasta este moştenirea multiplă.
- ✓ Când o clasă este derivată din două sau mai multe clase, clasele de bază se separă prin virgule.
- ✓ Moştenirea pe mai multe nivele are loc când o clasă este derivată dintr-o clasă de bază, care, la rândul ei, este derivată din altă clasă.
- ✓ Un membru protejat al unei clase poate fi direct accesibil de clasele derivate, dar imposibil de apelat din altă parte a programului. Acest tip de membru se foloseşte pentru a oferi membrilor derivaţi acces direct la membrii clasei de bază, în acelaşi timp protejându-i pe aceştia din urmă de restul programului.

CAPITOLUL 5
ACOMODAREA CU SUPRAPUNEREA FUNCŢIILOR ŞI OPERATORILOR

Prin crearea claselor în programele C++, se obţin propriile tipuri de date. În sensul cel mai simplu, un *tip* defineşte un set de valori de date ce pot fi memorate şi un set de operaţii ce se pot efectua cu aceste date. De exemplu, o variabilă de tip *int* poate memora valori întregi în domeniul -32 768, 32 767. Operaţiile ce pot fi realizate cu această variabilă sunt adunarea, scăderea, înmulţirea, împărţirea şi mai multe operaţii bit cu bit.

La crearea claselor, este de dorit ca unele operaţii să fie exprimate prin operatori. De exemplu, să presupunem că într-un program se foloseşte o variabilă a unei clase ce exprimă data sub forma zi, lună, an. Dacă se doreşte adunarea a 30 de zile la dată, sau scăderea a 15 zile din dată, operatorii plus şi minus pot produce un cod foarte inteligibil, de forma:

```
invoice_date = order_date + 30;
first_notice = invoice_date - 15;
```

Aveţi în vedere, totuşi, că variabilele membru ale clasei conţin câmpurile zi, lună şi an. Pentru ca operatorii plus şi minus să aibă sens, programul trebuie să definească operaţiile ce vor trebui efectuate la întâlnirea acestor operatori. Cu alte cuvinte, programul va efectua un anumit set de operaţii (probabil adunarea şi scăderea) când va întâlni operatorii plus şi minus folosiţi cu variabile de tip *int* sau *float*, şi alt set de operaţii când va întâlni aceiaşi operatori folosiţi cu variabilele clasei de date. **Suprapunerea operatorilor** este procesul de atribuire a două sau a mai multor operaţii aceluiaşi operator. Prin folosirea operatorilor supradefiniţi, textul programului poate deveni foarte natural şi uşor de înţeles. Acest capitol analizează în detaliu suprapunerea operatorilor şi a funcţiilor. Când veţi termina de citit capitolul, veţi constata că suprapunerea operatorilor este un concept uşor şi puternic, precum şi că aţi aflat următoarele:

- Ce reprezintă suprapunerea funcţiilor
- Când două funcţii distincte au acelaşi nume, cum determină C++ alegerea funcţiei ce va fi apelată
- Cum se îmbunătăţeşte claritatea programului prin suprapunerea funcţiilor

- Utilizarea parametrilor prestabiliţi
- Ce-i de făcut când se folosesc parametri, alţii decât cei prestabiliţi
- Cum se creează o bibliotecă de funcţii
- Ce reprezintă suprapunerea operatorilor
- Cum determină compilatorul C++ operaţia dorită
- Cum se supraîncarcă un operator
- Ce operatori permit suprapunerea în C++
- Cum se foloseşte un operator suprapus
- Cum se creează un operator suprapus cu acces la structurile membru ale unei clase

SĂ ÎNŢELEGEM SUPRAPUNEREA FUNCŢIILOR

Înainte de a examina suprapunerea operatorilor, este important de a înţelege suprapunerea funcţiilor. Aşa cum veţi vedea, ambele concepte sunt destul de asemănătoare şi implementate de compilatoarele C++. **Suprapunerea funcţiilor** este procesul de definire a două sau mai multor funcţii folosind acelaşi nume, care diferă numai prin tipul de date returnat sau prin numărul şi tipul parametrilor. Compilatorul va determina care funcţie trebuie apelată pe baza modului în care aceasta este folosită. De exemplu, următorul program, OVERLOAD.CPP, creează două funcţii denumite *show_message*. Prima funcţie nu are parametri, ci pur şi simplu afişează un mesaj. A doua funcţie primeşte ca parametru un şir de caractere:

```
#include <iostream.h>

void show_message(void)
 {
   cout << "Success with C++!" << endl;
 }

void show_message(char *message)
 {
   cout << message;
 }

void main(void)
 {
   show_message("My favorite book is: ");
   show_message();
 }
```


La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> OVERLOAD <ENTER>
My favorite book is: Success with C++!
```

După cum se poate vedea, primul apel de funcţie face referinţă la acea definiţie *showmessage* care suportă un şir de caractere drept parametru, în timp ce al doilea apel se referă la funcţia care afişează mesajul prestabilit. La compilare şi execuţie, compilatorul C++ va determina care funcţie trebuie apelată, pe baza numărului de parametri.



SĂ ÎNŢELEGEM SUPRAPUNEREA FUNCŢIILOR

Suprapunerea funcţiilor este procesul prin care se definesc în program două sau mai multe funcţii cu acelaşi nume. Funcţiile diferă numai prin numărul sau tipul parametrilor. La compilarea programului, C++ va determina univoc care din funcţii trebuie apelată, pe baza parametrilor fiecărei funcţii. În acest mod, programul va apela întotdeauna funcţia corectă.

***Observaţie:** Deşi suprapunerea funcţiilor poate fi un instrument foarte puternic şi oportun, folosirea în exces a procedeului poate duce, totuşi, la micşorarea clarităţii programului. Astfel, dacă definiţi mai multe versiuni ale aceleiaşi funcţii, estimaţi utilizarea suprapunerii funcţiilor pentru a vă asigura că programul respectiv este încă uşor de citit şi înţeles.*

ANALIZA ALTOR EXEMPLE

Aşa cum am văzut, compilatorul C++ determină ce funcţie trebuie apelată, în funcţie de parametrii transferaţi. Fie următorul program, INTARRAY.CPP, ce foloseşte funcţia *sum_values* pentru a calcula suma valorilor unui tablou de întregi:

```
#include <iostream.h>

long sum_array(int *array, int num_elements)
 {
   long sum = 0L;

   for (int i = 0; i < num_elements; i++)
     sum += array[i];

   return(sum);
 }

void main(void)
 {
```

```
  int array[5] = {1, 2, 3, 4, 5};

  cout << "The values sum to " << sum_array(array, 5) << endl;
}
```

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> INTARRAY <ENTER>
The values sum to 15
```

Să presupunem acum că dorim să însumăm valorile unui tablou de numere reale, definit în continuare:

```
float float_array[5] = {1.1, 2.2., 3.3, 4.4, 5.5};
```

Dacă se folosea limbajul C, era necesară crearea a două funcţii cu nume diferite pentru însumarea valorilor din cele două tablouri, cum se arată în programul C_ARRAYS.C de mai jos:

```
#include <stdio.h>

float sum_float_array(float *array, int num_elements)
{
  float sum = 0.0;
  int i;

  for (i = 0; i < num_elements; i++)
    sum += array[i];

  return(sum);
}

long sum_int_array(int *array, int num_elements)
{
  long sum = 0L;
  int i;

  for (i = 0; i < num_elements; i++)
    sum += array[i];

  return(sum);
}

void main(void)
{
  int int_array[5] = {1, 2, 3, 4, 5};
  float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

  printf("Values in int array: %ld\n",
         sum_int_array(int_array, 5));
  printf("Values in float array: %f\n",
         sum_float_array(float_array, 5));
}
```


După cum se poate vedea, programul defineşte funcţiile *sum_int_array* şi *sum_float_array* pentru a însuma valorile de tip *int* şi *float* din cele două tablouri. Deşi programul este corect, faptul că trebuie să creăm două funcţii distincte pentru fiecare tip este destul de incomod şi reduce claritatea programului. Folosind suprapunerea funcţiilor din C++, putem să folosim un singur nume de funcţie, *sum_array*, şi să transferăm drept parametri ai acestei funcţii tablouri de fiecare tip. Compilatorul va putea determina care funcţie trebuie apelată. Următorul program, SUMARRAY.CPP, realizează acest lucru.

```
#include <iostream.h>

float sum_array(float *array, int num_elements)
{
  float sum = 0.0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return(sum);
}

long sum_array(int *array, int num_elements)
{
  long sum = 0L;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return(sum);
}

void main(void)
{
  int int_array[5] = {1, 2, 3, 4, 5};
  float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

  cout << "Values in int array: " << sum_array(int_array, 5) <<
       endl;
  cout << "Values in float array: " << sum_array(float_array, 5) <<
       endl;
}
```

După cum se observă, programul foloseşte un singur nume de funcţie, *sum_array*, pentru tablouri de tip *int* şi *float*. Este important de reţinut că trebuie totuşi definite două funcţii, câte una pentru fiecare tip, dar programul poate folosi acelaşi nume de funcţie pentru fiecare tip de tablou.

În programul precedent, compilatorul C++ a putut determina, pe baza tipului de parametri, care din funcţiile *sum_array* să fie apelată. În programul următor, GETLINE.CPP, compilatorul va determina care funcţie *getline* să fie apelată, pe

baza numărului de parametri. În Capitolul 1 aţi învăţat că funcţia membru *getline* permite streamului de intrare *cin* să citească o linie de text de la tastatură sau de la dispozitivul standard de intrare:

```
cin.getline(line, sizeof(line));
```

Când folosiţi *getline*, puteţi de asemenea, specifica un caracter la care operaţia de intrare să se încheie. De exemplu, următorul apel de funcţie citeşte fie o linie de text, fie o parte a liniei de text, până la întâlnirea caracterului 'x' exclusiv:

```
cin.getline(line, sizeof(line), 'X');
```

Următorul program, GETLINE.CPP, foloseşte suprapunerea funcţiilor pentru a crea două funcţii ce suportă aceste două operaţii:

```
#include <iostream.h>

void getline(char *line, int size)
 {
   char letter;

   for (int i = 0; i < size; i++)
    if ((letter = cin.get()) == '\n')
      break;
    else
      line[i] = letter;

    line[i] = NULL;
 }

void getline(char *line, int size, char terminator)
 {
   char letter;

   for (int i = 0; i < size; i++)
    if ((letter = cin.get()) == '\n')
      break;
    else if (letter == terminator)
      break;
    else
      line[i] = letter;

    line[i] = NULL;
 }

void main(void)
 {
   char text[256];

   cout << "Type in a line of text: ";
   getline(text, sizeof(text));
   cout << "You typed: " << text << endl;
```



```
   cout << "Type text followed by X: ";
   getline(text, sizeof(text), 'X');
   cout << "You typed: " << text << endl;
 }
```

Aşa cum se poate vedea, în program se definesc două funcţii *getline* distincte. În timpul compilării, C++ determină corect funcţia ce trebuie apelată. Funcţia *getline* definită în acest program nu afectează funcţia membru *cin.getline*, datorită regulilor de vizibilitate discutate în Capitolul 9.

În sfârşit, următorul program, SHOWDATE.CPP, creează câteva funcţii denumite *show_date*. Prima dintre ele afişează data sistem curentă. A doua afişează data specificată într-un şir de caractere, iar a treia afişează data conţinută într-o structură:

```
#include <iostream.h>
#include <time.h>

void show_date(void)
 {
   time_t current_datetime;

   time(&current_datetime);

   cout << "Current date: " << ctime(&current_datetime);
 }

void show_date(char *date)
 {
   cout << "String date: " << date << endl;
 }

struct Date {
  int month;
  int day;
  int year;
 };

void show_date(struct Date date)
 {
   cout << "Structure date: ";

   switch (date.month) {
     case  1: cout << "January ";
              break;
     case  2: cout << "February ";
              break;
     case  3: cout << "March ";
              break;
```

```
    case  4: cout << "April ";
             break;
    case  5: cout << "May ";
             break;
    case  6: cout << "June ";
             break;
    case  7: cout << "July ";
             break;
    case  8: cout << "August ";
             break;
    case  9: cout << "September ";
             break;
    case 10: cout << "October ";
             break;
    case 11: cout << "November ";
             break;
    case 12: cout << "December ";
             break;
  };

  cout << date.day << ", " << date.year << endl;
}

void main(void)
{
  struct Date date = {9, 30, 1994};
  char datestr[9];

  show_date(); // Show default

  show_date(_strdate(datestr));

  show_date(date);
}
```

După cum se observă, programul foloseşte 3 funcţii diferite, toate având numele *show_date*. În funcţie de parametrii pe care programul îi transferă la apelul funcţiilor, compilatorul C++ va determina care din funcţii va fi apelată. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SHOWDATE  <ENTER>
Current date: Wed Nov 24 19:30:03 1993
String date: 11/24/93
Structure date: September 30, 1994
```

Observaţie: *Pentru simplitate, programul precedent a folosit o instrucţiune case pentru a afişa numele lunilor conţinute în câmpul date.month. Printr-o schimbare simplă de program se poate folosi un tablou de pointeri la şiruri de caractere pentru afişarea lunilor, ca mai jos:*


```
void show_date(struct Date date)
{
  char *months[] = { "January", "February", "March", "April",
                     "May", "June", "July", "August", "September",
                     "October", "November", "December" };

  cout << "Structure date: ";

  cout << months[date.month] << " " << date.day << ", " <<
          date.year << endl;
}
```

Aşa cum se observă, prin definirea în acest mod a tabloului de şiruri de caractere, se reduce numărul de instrucţiuni din funcţie, ceea ce duce la îmbunătăţirea clarităţii codului.

FOLOSIREA PARAMETRILOR PRESTABILIŢI

Pe lângă posibilitatea de a suprapune funcţiile, C++ permite, de asemenea, specificarea *valorilor parametrilor impliciţi* ai unei funcţii. În acest fel, dacă programul apelează o anumită funcţie şi omite unul sau mai mulţi parametri, funcţia va folosi valori prestabilite. Pentru a înţelege mai bine cum funcţionează mecanismul parametrilor prestabiliţi, să considerăm programul DEF_PAR.CPP care defineşte funcţia *show_values*. Funcţia are 3 parametri, iar dacă este apelată fără ca toţi parametrii să fie specificaţi, se vor folosi valorile prestabilite acolo unde este necesar:

```
#include <iostream.h>

void show_values(int a = 1, int b = 2, int c = 3)
{
  cout << a << ' ' << b << ' ' << c << endl;
}

void main(void)
{
  show_values();
  show_values(1001);
  show_values(1001, 2002);
  show_values(1001, 2002, 3003);
}
```

După cum se vede, programul apelează funcţia de patru ori, folosind patru combinaţii diferite de parametri. La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> DEF_PAR  <ENTER>
1 2 3
1001 2 3
```

```
1001 2002 3
1001 2002 3003
```

Se observă că, atunci când programul nu specifică valori pentru parametri, funcţia foloseşte parametrii prestabiliţi.

Când se omite un parametru al unei funcţii care acceptă parametri prestabiliţi, trebuie omise valorile tuturor celorlalţi parametri care urmează. De exemplu, în programul precedent, nu se putea preciza valoarea parametrului *a*, pentru a se omite valoarea pentru parametrul *b*, iar apoi să se specifice valoarea pentru parametrul *c*, ca mai jos:

```
show_values(1001, , 3003);
```

Eroare de sintaxă la omiterea parametrului din mijloc

În cazul precedent, dacă programul omite valoarea pentru parametrul *b*, el trebuie să omită şi valoarea pentru parametrul *c*.

Anterior aţi creat, în acest capitol, două versiuni ale funcţiei *getline*. După cum probabil vă amintiţi, prima funcţie citea o linie de text de la tastatură până la întâlnirea caracterului CR exclusiv. A doua funcţie permitea programului să specifice un caracter de încheiere. Următorul program, DEF_LINE.CPP, modifică programul anterior pentru a folosi o valoare prestabilită de parametru pentru caracterul de încheiere. În acest mod, programul poate atinge acelaşi scop folosind o singură funcţie:

```
#include <iostream.h>

void getline(char *line, int size, char terminator = '\n')
 {
   char letter;

   for (int i = 0; i < size; i++)
    if ((letter = cin.get()) == terminator)
      break;
    else
      line[i] = letter;

    line[i] = NULL;
 }

void main(void)
 {
   char text[256];

   cout << "Type in a line of text: ";
   getline(text, sizeof(text));
   cout << "You typed: " << text << endl;
   cout << "Type text followed by X: ";
   getline(text, sizeof(text), 'X');
   cout << "You typed: " << text << endl;
 }
```


Aşa cum se observă, programul foloseşte acum o singură funcţie *getline*. Dacă programul va apela funcţia *getline* fără a se specifica caracterul de încheiere, va fi folosit, în mod prestabilit, caracterul '\n'. În cazul specificării caracterului de încheiere drept parametru, programul va citi până când se va umple bufferul sau până la întâlnirea caracterului respectiv. În acest mod, programul va putea citi caractere şi după întâlnirea caracterului '\n'. Astfel se poate folosi funcţia getline pentru a citi de la tastatură caractere care includ şi caracterul "\n".

Este interesant de remarcat că, dacă examinaţi definiţia clasei *istream* în fişierul antet folosit de compilatorul C++ pentru funcţii stream, veţi găsi pentru funcţia *getline* un prototip de forma:

```
istream _FAR & _Cdecl getline(unsigned char _FAR *, int, char = '\n');
```

Se observă că funcţia specifică caracterul sfârşit de linie drept valoare prestabilită pentru parametrul ce indică încheierea citirii.



SĂ ÎNŢELEGEM PARAMETRII PRESTABILIŢI

Când scrieţi funcţii în C++, este posibil ca majoritatea programelor să folosească aceleaşi valori pentru anumiţi parametri. Pentru simplificarea folosirii acestor funcţii, se pot specifica valori prestabilite pentru parametri. O valoare prestabilită este specificată în antetul funcţiei, fiind precedată de semnul egal. De exemplu, următoarea funcţie *getline* specifică caracterul *newline* drept parametru prestabilit:

```
void getline (char * text, int size, char terminator = '\n')
```

Dacă programul apelează funcţia *getline* numai cu doi parametri, al treilea parametru va avea valoarea prestabilită:

```
getline (line, sizeof (line));
```

Dacă programul specifică valoarea de terminare, valoarea implicită va fi neglijată:

```
getline (line, sizeof (line), 'x');
```

O funcţie poate avea mai mulţi parametri impliciţi. De exemplu, funcţia *get_tax* foloseşte doi parametri prestabiliţi:

```
float get_tax (float amount, float sales_tax = 0.06,
               float state_tax = 0.03);
```

> În acest caz, programul poate omite valorile pentru parametrii *sales_tax* şi *state_tax*. Când programul omite valoarea pentru un parametru, el trebuie să omită valorile pentru toţi parametrii care urmează. În cazul de mai sus, dacă programul omite valoarea pentru parametrul *sales_tax*, el trebuie să omită valoarea şi pentru parametrul *state_tax*.

CUM SĂ CREĂM PROPRIILE BIBLIOTECI DE FUNCŢII

Pe măsură ce numărul funcţiilor creşte, vom dori să creăm propriile noastre *biblioteci de funcţii*, în care programele să caute funcţiile de care au nevoie. Pentru a facilita folosirea funcţiilor, trebuie creat un fişier antet, care să conţină prototipul fiecărei funcţii. La funcţiile care suportă parametri prestabiliţi, valorile corespunzătoare trebuie specificate în fişierul antet.

SĂ ÎNŢELEGEM SUPRAPUNEREA OPERATORILOR

Suprapunerea operatorilor este procesul de atribuire a două sau mai multor operaţii aceluiaşi operator. În funcţie de modul de utilizare a operatorului, compilatorul C++ va determina operaţia care se va realiza. Pentru a înţelege mai bine acest concept, să examinăm un program care creează o clasă *stack* cu care se pot depune valori în memorie (*push*) sau se pot scoate (*pop*) din memorie. Clasa *stack* (stivă) are următoarea formă:

```
class stack {
public:
  stack(int size);
  int push(int value);
  int pop(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  int *storage;    // Stack data buffer
  int elements;    // Number of values in the stack
  int isempty;     // True when stack is empty
  int isfull;      // True when stack is full
  int stack_size;  // Number of values the stack can store
};
```

Următorul program, STACK.CPP, creează un obiect de tipul *stack* ce poate conţine 64 de valori. Programul depune apoi valori în stivă, până când aceasta se umple. Ulterior, programul scoate valorile din stivă, una câte una:

```
#include <iostream.h>

class stack {
```



```
public:
  stack(int size);
  int push(int value);
  int pop(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  int *storage;            // Stack data buffer
  int elements;            // Number of values in the stack
  int isempty;             // True when stack is empty
  int isfull;              // True when stack is full
  int stack_size;          // Number of values the stack can store
};

stack::stack(int size)
 {
   storage = new int[size];  // Allocate memory for the stack
   elements = 0;
   isempty = 1;
   isfull = 0;
   stack_size = size;
 }

int stack::pop(void)
 {
   if (is_empty())
     return(0);
   else
     {
       if (--elements == 0)
         isempty = 1;

       isfull = 0;

       return(storage[elements]);
     }
 }

int stack::push(int value)
 {
   if (is_full())
     return(0);
   else
     {
       isempty = 0;
       storage[elements++] = value;

       if (elements == stack_size)
         isfull = 1;

       return(value);
```

```
    }
}

void main(void)
{
  stack fifo(64);

  int i;

  for (i = 0; ! fifo.is_full(); i++)
    fifo.push(i);

  while (! fifo.is_empty())
    cout << fifo.pop() << endl;
}
```

După cum se poate vedea, programul foloseşte funcţiile membru *push* şi *pop* pentru a depune, respectiv pentru a scoate elemente din stivă. Pentru a facilita citirea şi înţelegerea programelor, se pot supradefini operatorii + şi (– –) pentru clasa *stack*. De exemplu, operatorul + s-ar putea folosi în locul funcţiei *push*, iar operatorul – – în locul funcţiei *pop*:

```
pushed_value = fifo + 54;
fifo = -fifo;
```

La suprapunerea unui operator, se specifică funcţia pe care programul o execută când compilatorul C++ întâlneşte operatorul respectiv. De exemplu, când C++ întâlneşte operatorul + cu obiectele de tipul *stack*, compilatorul va insera cod pentru apelarea funcţiei şi nu va realiza o adunare efectivă. Pentru a defini o funcţie operator, se defineşte o funcţie obişnuită, exceptând includerea cuvântului-cheie *operator*. Următoarea funcţie, de exemplu, suprapune operatorul +:

Tip valoare returnată — Nume clasă

```
int stack::operator +(int value)
{
  if (is_full())
    return(0);
  else
    {
      isempty = 0;
      storage[elements++] = value;

      if (elements == stack_size)
        isfull = 1;

      return(value);
    }
}
```

Parametru — Operator — Cuvânt cheie **operator**

Analog, următoarea funcţie supradefineşte operatorul – –:

```
int stack::operator -(void)
{
  if (is_empty())
    return(0);
  else
    {
      if (-elements == 0)
        isempty = 1;

      isfull = 0;

      return(storage[elements]);
    }
}
```

La definirea operatorilor unei clase, aceştia trebuie incluşi în definiţia clasei, după cum se arată mai jos:

```
class stack {
public:
  stack(int size);
  int operator +(int value);
  int operator --(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  int *storage;   // Stack data buffer
  int elements;   // Number of values in the stack
  int isempty;    // True when stack is empty
  int isfull;     // True when stack is full
  int stack_size; // Number of values the stack can store
};
```

Operatori clasă de tip *public*

Următorul program, STACKOVR.CPP foloseşte operatorii + şi – – pentru a plasa şi a scoate valori din stivă:

```
#include <iostream.h>

class stack {
public:
  stack(int size);
  int operator +(int value);
  int operator -(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  int *storage;              // Stack data buffer
  int elements;              // Number of values in the stack
```

```
  int isempty;             // True when stack is empty
  int isfull;              // True when stack is full
  int stack_size;          // Number of values the stack can store
};

stack::stack(int size)
 {
   storage = new int[size];  // Allocate memory for the stack
   elements = 0;
   isempty = 1;
   isfull = 0;
   stack_size = size;
 }

int stack::operator --(void)
 {
   if (is_empty())
     return(0);
   else
     {
       if (--elements == 0)
         isempty = 1;

       isfull = 0;

       return(storage[elements]);
     }
 }

int stack::operator +(int value)
 {
   if (is_full())
     return(0);
   else
     {
       isempty = 0;
       storage[elements++] = value;

       if (elements == stack_size)
         isfull = 1;

       return(value);
     }
 }

void main(void)
 {
   stack fifo(64);

   int i;
```



```
   for (i = 0; ! fifo.is_full(); i++)
     fifo + i;

   while (! fifo.is_empty())
     cout << --fifo << endl;
 }
```

Aşa cum se observă, programul foloseşte următoarea instrucţiune pentru a depune valori în stivă:

```
fifo + i;
```

Pentru afişarea valorii unui element scos din stivă, se foloseşte instrucţiunea:

```
cout << --fifo << endl;
```



SĂ ÎNŢELEGEM SUPRAPUNEREA OPERATORILOR

Suprapunerea operatorilor este procesul de atribuire a două sau mai multor operaţii aceluiaşi operator. Prin suprapunerea unor operatori, programele pot exprima operaţiile din interiorul claselor într-un mod mai natural. De exemplu, operatorul + poate fi folosit pentru adunare obişnuită, pentru a insera zile într-o dată de tip structură, sau pentru a concatena două şiruri de caractere, cum se arată mai jos:

```
value = some_value + 100;

expiration_date = sales_date + 30;      // date class

new_string = "Success " + "with C++!";  // string class
```

Pentru a suprapune un operator, se defineşte o funcţie pe care compilatorul C++ o apelează ori de câte ori găseşte acele tipuri de date pentru care a fost definit operatorul. Definiţia funcţiei operator este asemănătoare unei funcţii obişnuite, cu excepţia faptului că trebuie inclus cuvântul cheie *operator* în antetul funcţiei.

REGULI PENTRU SUPRAPUNEREA OPERATORILOR

La suprapunerea unui operator, se specifică de fapt două tipuri de date pentru care se doreşte o tratare diferită a operaţiei respective. Astfel, un operator poate fi folosit numai în modul în care programul îl utilizează în mod obişnuit. De exemplu, nu se poate folosi un operator unar, cum este -- (operator de decrementare), pentru a realiza o operaţie pe două valori:

```
result = a - b;   // Syntax error
```

În general, C++ permite suprapunerea tuturor operatorilor, cu excepţia celor prezentaţi în Tabela 5.1:

Operator	Scop	Exemplu
.	membru al unei clase sau al unei structuri	cin.get ()
.*	pointer la membru	object.*member
::	operator de rezoluţie globală	class_name::member
?:	operatorul condiţional	c = (a > b)?a:b

Tabela 5.1. *Operatorii ce nu pot fi supradefiniţi în C++.*

SUPRAPUNEREA OPERATORILOR I/O

În programul precedent am suprapus operatorii + şi - - pentru a-i folosi cu obiecte de tip stivă. În funcţie de modul de utilizare a claselor, pot apare şi situaţii când se doreşte suprapunerea operatorilor de inserţie (<<) şi extracţie (>>). Să considerăm următorul program, BOOKONE.CPP, care foloseşte clasa *book* pentru a memora informaţii despre o anumită carte:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class book {
public:
  book(char *title, char *author, char *publisher, float price);
  void show_book(void);
private:
  char title[64];
  char author[64];
  char publisher[64];
  float price;
};

book::book(char *title, char *author, char *publisher, float price)
 {
   strcpy(book::title, title);
   strcpy(book::author, author);
   strcpy(book::publisher, publisher);
   book::price = price;
 }

void book::show_book(void)
 {
```



```
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Publisher: " << publisher << endl;
   cout << setprecision(2) << "Price: " << price << endl;
 }

void main(void)
 {
   book computer_book("Success with C++", "Kris Jamsa",
                      "Jamsa Press", 29.95);

   computer_book.show_book();
 }
```

După cum se observă, programul foloseşte funcţia membru *show_book* pentru a afişa informaţia despre carte. În cadrul funcţiei, se foloseşte operatorul de inserţie cu *cout*. Programul ar putea elimina funcţia membru *show_book* prin suprapunerea operatorului de extracţie (<<) pentru a-l utiliza la obiecte de tipul *book*. În acest mod, programul va putea afişa informaţia despre carte folosind operatorul de extracţie astfel:

```
cout << computer_book;
```

Pentru a suprapune un operator, programul trebuie să creeze o funcţie care se execută de fiecare dată când operatorul este folosit cu clasa respectivă. Următoarele instrucţiuni suprapun operatorul de extracţie pentru a fi folosit cu obiecte de tipul *book*:

```
ostream& operator <<(ostream&stream, book bookinfo)
 {
   cout << "Title: " << bookinfo.title << endl;
   cout << "Author: " << bookinfo.author << endl;
   cout << "Publisher: " << bookinfo.publisher << endl;
   cout << setprecision(2) << "Price: " << bookinfo.price << endl;
   return(stream);
 }
```

După cum se vede, definiţia operatorului este în mare parte similară funcţiei *show_book* folosită anterior, prin aceea că diferite câmpuri ale obiectului *book* sunt scrise în *cout*.

Cea mai confuză parte a definiţiei este, de departe, antetul funcţiei:

```
ostream& operator <<(ostream&stream, book bookinfo)
```

Pentru a înţelege această instrucţiune, să ne reamintim că, la suprapunerea unui operator se defineşte o funcţie pe care programul o execută când este întâlnit operatorul. În acest caz, funcţia returnează o referinţă la un obiect de tipul *ostream*. Parametrii operaţiei sunt un obiect *ostream* (probabil *cout*) şi un obiect de tipul *book*:

Tip valoare returnată

```
ostream& operator <<(ostream& stream, book bookinfo)
```

Parametri funcţie

Pentru a înţelege parametrii funcţiei, să ne gândim la operanzii care apar în partea stângă şi în partea dreaptă a operatorului:

Obiect *ostream*

```
cout << computer_book;
```

Obiect *book*

În sfârşit, după ce funcţia inserează informaţiile în stream-urile de ieşire, se foloseşte instrucţiunea *return* pentru a returna stream-ul actualizat:

```
return(stream);
```

Dacă nu înţelegeţi prea bine referinţele la stream-urile I/O, nu vă faceţi griji. În Capitolul 10 vom discuta referinţele C++ în detaliu.

Următorul program, BOOKTWO.CPP, foloseşte suprapunerea operatorilor pentru a afişa informaţia despre respectivul obiect de tip *book*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class book {
public:
  book(char *title, char *author, char *publisher, float price);
  friend ostream& operator <<(ostream& stream, book bookinfo);
private:
  char title[64];
  char author[64];
  char publisher[64];
  float price;
};

book::book(char *title, char *author, char *publisher, float price)
 {
   strcpy(book::title, title);
   strcpy(book::author, author);
   strcpy(book::publisher, publisher);
   book::price = price;
 }

ostream& operator <<(ostream&stream, book bookinfo)
 {
   cout << "Title: " << bookinfo.title << endl;
   cout << "Author: " << bookinfo.author << endl;
   cout << "Publisher: " << bookinfo.publisher << endl;
```



```
  cout << setprecision(2) << "Price: " << bookinfo.price << endl;
  return(stream);
}

void main(void)
{
  book computer_book("Success with C++", "Kris Jamsa",
    "Jamsa Press", 29.95);

  cout << "Book Information" << endl;
  cout << computer_book;
}
```

După cum se observă, programul afişează informaţii despre obiectul *book* folosind operatorul de inserţie. De notat că programul foloseşte, de asemenea, operatorul de inserţie pentru a scrie un mesaj la *cout* imediat înainte de a afişa variabilele membru ale obiectului.

```
cout << "Book Information" << endl;
cout << computer_book;
```

Cele două linii s-ar fi putut chiar combina sub forma următoare:

```
cout << "Book Information" << endl << computer_book;
```

Reţineţi că operatorul suprapus este folosit numai cu obiecte de tip *book*. Astfel, prima instrucţiune foloseşte operatorul de inserţie standard, în timp ce a doua instrucţiune foloseşte acelaşi operator suprapus. În acest fel, un operator suprapus este similar unei funcţii suprapuse. Să mai observăm că, în cadrul definiţiei clasei, operatorul este suprapus ca o funcţie *friend*:

```
class book {
public:
  book(char *title, char *author, char *publisher, float price);
  friend ostream& operator <<(ostream& stream, book bookinfo);
private:
  char title[64];
  char author[64];
  char publisher[64];
  float price;
};
```

Operator declarat ca friend

Declarând operatorul de inserţie ca o funcţie *friend*, clasa îi acordă dreptul de acces la membrii săi de tip *private*.

ANALIZA ALTOR EXEMPLE

Cu cât programele folosesc mai mult suprapunerea operatorilor, cu atât mai mult veţi aprecia puterea şi facilităţile acestui concept. De aceea, în această secţiune vom mai examina câteva programe ce supradefinesc operatorii pentru diferite

tipuri de clase. Experimentaţi cât mai mult aceste programe. Veţi vedea că suprapunerea operatorilor nu este dificilă, o dată ce aţi practicat-o de câteva ori.

Următorul program, DATE_OPS.CPP, suprapune operatorii de inserţie şi extracţie astfel încât să se poată realiza operaţii de intrare/ieşire pe structura *Date*:

```
#include <iostream.h>
#include <string.h>

class Date {
  public:
    Date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, Date date);
    friend istream& operator >>(istream& stream, Date *date);
  private:
    int month;
    int day;
    int year;
};

Date::Date(int month, int day, int year)
 {
   Date::month = month;
   Date::day = day;
   Date::year = year;
 }

ostream& operator <<(ostream& stream, Date date)
 {
   stream << date.month << '/' << date.day << '/' << date.year <<
           endl;

   return(stream);
 }

istream& operator >>(istream& stream, Date *date)
 {
   stream >> date->month;
   stream >> date->day;
   stream >> date->year;

   return(stream);
 }

void main(void)
 {
   Date christmas(12, 25, 94);
   Date birthday(9, 30, 94);

   cout << "Christmas is " << christmas;
```



```
   cout << "Type in your birthday mm dd yy: ";
   cin >> &birthday;
   cout << "You typed: " << birthday;
 }
```

Se observă că, în cadrul clasei *Date*, operatorii de inserţie şi extracţie sunt declaraţi drept *friend*, pentru a avea acces la variabilele membru de tip private:

```
class Date {
  public:
    Date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, Date date);
    friend istream& operator >>(istream& stream, Date *date);
  private:
    int month;
    int day;
    int year;
};
```

Declararea funcţiilor operator ca **friend**

Funcţiile operator suprapuse returnează o referinţă la un stream, primind drept parametri stream-ul şi clasa:

Returnează o referinţă la stream
Parametru stream
Structură date

```
ostream& operator <<(ostream& stream, Date date)
 {
   stream << date.month << '/' << date.day << '/' <<
             date.year << endl;
   return(stream);
 }
```

Returnează o referinţă la stream
Parametru stream
Pointer la structură de date

```
istream& operator >>(istream& stream, Date *date)
 {
   stream >> date->month;
   stream >> date->day;
   stream >> date->year;

   return(stream);
 }
```

Deoarece operatorul de extracţie suprapus atribuie valori variabilelor clasei, funcţia trebuie să lucreze cu un pointer la clasă. În cazurile când în programe se lucrează cu clase sau structuri ce conţin mai multe valori, cum ar fi o dată, apar situaţii când este nevoie de a testa egalitatea a două clase sau două structuri. Următorul program TEST_OPS.CPP, suprapune operatorul de egalitate (==) şi inegalitate (!=) pentru lucrul cu clase de tipul *Date*. Folosind aceşti operatori, programul poate determina rapid dacă două clase conţin sau nu aceleaşi valori.

```
#include <iostream.h>
#include <string.h>

class Date {
  public:
    Date(int month, int day, int year);
    friend int operator ==(Date date_one, Date date_two);
    friend int operator !=(Date date_one, Date date_two);
  private:
    int month;
    int day;
    int year;
};

Date::Date(int month, int day, int year)
 {
   Date::month = month;
   Date::day = day;
   Date::year = year;
 }

int operator ==(Date date_one, Date date_two)
 {
   if (date_one.day != date_two.day)
     return(0);
   else if (date_one.month != date_two.month)
     return(0);
   else if (date_one.year != date_two.year)
     return(0);
   return(1);
 }

int operator !=(Date date_one, Date date_two)
 {
   if (date_one.day != date_two.day)
     return(1);
   else if (date_one.month != date_two.month)
     return(1);
   else if (date_one.year != date_two.year)
     return(1);
   return(0);
 }

void main(void)
 {
   Date christmas(12, 25, 94);
   Date santas_day(12, 25, 94);
   Date birthday(9, 30, 94);

   if (christmas == santas_day)
     cout << "Christmas and Santa's day are the same" << endl;
```



```
   if (christmas != birthday)
     cout << "My birthday is not Christmas" << endl;
 }
```

Prin suprapunerea operatorilor == şi !=, programul devine mult mai sugestiv. Alternativa testării egalităţii a două clase de tipul *Date* ar fi putut fi:

```
if ((christmas.day) == birthday.day) &&
  (christmas.month == birthday.month) &&
  (christmas.year == birthday.year))
 cout << "My birthday is on Christmas!" << endl;
```

În acest caz, nu numai că secvenţa de cod devine mai dificil de înţeles, dar instrucţiunile suplimentare măresc posibilitatea apariţiei unei erori în program. Examinând programele din această carte, veţi observa cum suprapunerea operatorilor îmbunătăţeşte claritatea programelor şi capacitatea de înţelegere a acestora de către programator.

REZUMAT

Suprapunerea operatorilor este procesul atribuirii a două sau mai multor operaţii aceleiaşi funcţii. În acest capitol ambele concepte s-au examinat în detaliu. Înainte de a trece la Capitolul 6, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Suprapunerea funcţiilor este procesul de definire a două sau mai multor funcţii cu acelaşi nume, dar care diferă prin tipul rezultatului sau al parametrilor.
- ✓ Compilatorul C++ va determina apelarea unei anumite funcţii pe baza tipului rezultatului sau al parametrilor.
- ✓ Suprapunerea funcţiilor permite îmbunătăţirea clarităţii programului, prin folosirea unor funcţii cu acelaşi nume pentru aceleaşi operaţie, funcţii care diferă numai prin tipul de date folosit.
- ✓ Pentru a mări flexibilitatea folosirii funcţiilor, compilatorul C++ suportă parametri prestabiliţi. Dacă la apelul funcţiilor nu se specifică toţi parametrii actuali, atunci C++ îi va substitui cu valorile prestabilite.
- ✓ Dacă la apelul unei funcţii se omite un parametru, toţi parametrii care urmează trebuie omişi. Cu alte cuvinte, nu se poate specifica o valoare pentru parametrul 1, urmată de valoarea prestabilită a parametrului 2 şi din nou de o valoare a parametrului 3.
- ✓ Pentru a crea o bibliotecă de funcţii, se creează un fişier antet ce conţine prototipurile fiecărei funcţii. Dacă funcţiile suportă parametri prestabiliţi, aceştia se specifică în fişierul prototip.
- ✓ Suprapunerea operatorilor este atribuirea a două sau mai multor operaţii aceluiaşi operator. În timpul compilării, C++ deter-

mină operaţia ce trebuie efectuată, pe baza modului de utilizare a operanzilor.

- ✓ Pentru a suprapune un operator, programul specifică o funcţie pe care compilatorul o apelează când întâlneşte operatorul.
- ✓ Compilatorul C++ permite suprapunerea majorităţii operatorilor, excepţie făcând:

Operator	Nume
.	membru al clasei
.*	pointer la membru la clasei
::	rezoluţie globală
?:	atribuire condiţională

- ✓ Dacă un operator este suprapus, el trebuie folosit într-o manieră consecventă faţă de modul normal de utilizare a acestuia.
- ✓ La suprapunerea unui operator pentru o clasă, de obicei acesta se declară ca o funcţie *friend*, pentru a-i asigura accesul la membrii acelei clase.

Capitolul 6
Acomodarea cu şabloanele

În Capitolul 5 aţi învăţat cum suprapunerea funcţiilor şi operatorilor poate facilita înţelegerea programelor, prin folosirea unor nume de funcţii sugestive. De exemplu, să presupunem că un program are de calculat valoarea medie a unor tablouri de tip *int* sau *float*. Prin suprapunerea numelui de funcţie *average-value*, programul poate folosi două funcţii cu acelaşi nume, una pentru valorile întregi şi alta pentru valorile reale. Dacă alţi programatori vor citi programul, vor înţelege mai uşor scopul funcţiei *average_value*, decât dacă ar fi existat două funcţii cu nume diferite, ca *f_average value* şi *i_average_value* (cum ar fi fost necesar în C). Din păcate, cu toate că facilitatea de a folosi acelaşi nume de funcţie este foarte convenabilă, codul trebuie scris pentru fiecare funcţie în parte. În majoritatea cazurilor, singura diferenţă între astfel de funcţii este tipul datelor cu care operează funcţiile.

Acest capitol tratează funcţiile-şablon în C++, care permit definirea unui şablon (template) sau a unei schiţe ("blueprint") pentru definiţiile de funcţii. Folosind aceste şabloane, programele pot direcţiona compilatorul C++ pentru a crea în mod automat codul unei funcţii pentru diferite tipuri de variabile. De exemplu, un program ar putea crea un şablon *average_value*. Folosind ăcest şablon, programul va putea crea funcţii pentru tablouri de tip *int*, *float* şi altele. Pe lângă şabloane de funcţii, acest capitol tratează şi şabloane de clase, ceea ce vă permite să definiţi clase ale căror membri diferă numai prin tip.

La sfârşitul acestui capitol, veţi şti următoarele:

- Ce este un şablon şi la ce foloseşte
- C++ suportă şabloanele de funcţii şi clase
- Cum se creează un şablon de funcţie
- Cum se creează ulterior o funcţie ce returnează sau admite parametri de un anumit tip
- Cum atribuie compilatorul tipurile, dacă şablonul de funcţie admite mai multe tipuri de parametri
- Cum se creează un şablon de clasă
- Cum se creează obiecte folosind un şablon de clasă

- Cum atribuie compilatorul tipurile, dacă şablonul de clasă admite mai multe tipuri de parametri
- Cum realizează compilatorul substituţia de tipuri, dacă un şablon de clasă foloseşte alt şablon

CREAREA PRIMULUI ŞABLON DE FUNCŢIE

Un *şablon de funcţie* este un tipar din care compilatorul C++ construieşte funcţii pentru diferite tipuri de variabile. De exemplu, următoarele instrucţiuni creează un şablon pentru funcţia *average_value*.

```
template<class T> T average_value(T *array, int num_elements)
{
  T sum = 0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return (sum / num_elements);
}
```

Înainte de a examina codul funcţiei şablon, priviţi cu atenţie la prima linie:

```
template<class T> T average_value(T *array, int num_elements)
```

Cuvântul cheie *template* informează compilatorul C++ că instrucţiunile care urmează definesc un şablon. Aşa cum aţi citit, şabloanele de funcţii permit programelor să specifice instrucţiuni pentru funcţii care diferă numai prin tip. Simbolul *<class T>* care urmează după cuvântul-cheie *template* specifică un simbol care reprezintă tipul funcţiei sau tipul parametrilor funcţiei. În acest exemplu, simbolul tipului este T. Restul liniei indică rezultatul funcţiei şi tipul parametrilor, ca la o declaraţie de funcţie obişnuită.

Să presupunem, de exemplu, că fiecare apariţie a lui T va fi substituită prin tipul *float*. Antetul de funcţie va fi atunci:

```
float average_value(float *array, int num_elements)
```

Analog, dacă T va fi substituit cu *int*, antetul de funcţie devine:

```
int average_value(int *array, int num_elements)
```

Observaţi în corpul funcţiei instrucţiunea care foloseşte simbolul T pentru a declara variabila *sum*:

```
T sum = 0;
```

Când compilatorul C++ substituie pe T cu numele tipului în antetul de funcţie, va substitui şi simbolul specificat în corpul funcţiei. Pentru ca C++ să cunoască

tipul funcţiilor ce urmează a fi create, se pot specifica prototipuri de funcţii imediat după definiţia şablonului, ca mai jos:

```
int average_value(int *, int);

float average_value(float *, int);
```

În acest caz, compilatorul C++ va crea automat funcţii pentru tipul *int* şi *float*. După cum se vede în continuare, C++ foloseşte tipul primului parametru pentru a substitui pe T:

```
int average_value(int *, int);

  template<class T> T average_value(T *array, int num_elements)
  {
    T sum = 0;

    for (int i = 0; i < num_elements; i++)
      sum += array[i];

    return (sum / num_elements);
  }

     int average_value(int *array, int num_elements)
     {
       int sum = 0;

       for (int i = 0; i < num_elements; i++)
         sum += array[i];

       return (sum / num_elements);
     }
```

Pe lângă folosirea prototipurilor, programul poate folosi direct funcţia, transferând acesteia parametrii ai căror tipuri determină tipurile corespunzătoare de funcţii. Următorul program, TEMP_AVG.CPP foloseşte şablonul *average_value*:

```
#include <iostream.h>

template<class T> T average_value(T *array, int num_elements)
{
  T sum = 0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return (sum / num_elements);
}

int average_value(int *, int);
```

```
float average_value(float *, int);

void main(void)
 {
   int values[] = { 1, 2, 3, 4, 5 };
   float prices[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

   cout << "Average of integer values is " <<
           average_value(values, 5) << endl;
   cout << "Average of floating-point values is " <<
           average_value(prices, 5) << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> TEMP_AVG <ENTER>
Average of integer values is 3
Average of floating-point values is 3.3
```

Experimentaţi acest program cu răbdare, îndepărtând mai întâi prototipurile de funcţii. La o nouă compilare a programului, C++ va determina corect tipurile pentru şablon, în funcţie de parametrii transferaţi funcţiei. În al doilea rând, se poate adăuga la program următoarea declaraţie de tablou:

```
long distances[] = { 1000000L, 2000000L, 3000000L };
```

Pentru a determina valoarea medie a tabloului, folosiţi instrucţiunea:

```
cout << "Average of long values is " <<
        average_value(distances, 3) << endl;
```

Observaţie: *Funcţia şablon* average_value *concretizată pentru tipul* int *funcţionează în programul precedent deoarece suma valorilor nu depăşeşte suma variabilelor de tip* int*. Dacă se foloseşte, însă, următorul tablou, funcţia va furniza un rezultat eronat:*

```
int values[] = {1000, 5000, 10000, 15000, 20000};
```

Pentru a rezolva eroarea de depăşire, tipul variabilei *sum* trebuie să fie *long*:

```
long sum = 0;
```

Din păcate, această declaraţie poate crea probleme pentru alte tipuri de date, cum ar fi un tablou de tip *float*. Tot în acest capitol veţi învăţa cum se specifică mai multe tipuri de parametri în funcţiile şablon, pentru situaţii ca acestea.

SĂ ÎNŢELEGEM ŞABLOANELE DE FUNCŢII



Un *şablon de funcţie* este un tipar după care compilatorul poate construi automat diverse funcţii. Şabloanele de funcţii sunt ideale pentru funcţiile care diferă numai prin tip. Pentru crearea unui şablon de funcţie, se foloseşte cuvântul-cheie *template*, ca mai jos:

```
template<class T> T max_value(T *array, int num_elements)
 {
   T max = array[0];

   for (int i = 0; i < num_elements; i++)
     if (max < array[i])
      max = array[i];

   return(max);
 }
```

În acest exemplu, se creează un şablon pentru funcţii cu numele *max_value*. Construcţia *<class T>* specifică un simbol (aici este *T*) pe care compilatorul îl va substitui cu un tip concret de date. Pentru a indica compilatorului tipul dorit, se pot specifica în program prototipuri de funcţii similare celor de mai jos:

```
float max_value(float *, int);

int max_value(int *, int);
```

În timpul compilării, simbolul T va fi înlocuit cu primul tip de parametri creându-se funcţii suprapuse pentru tipul respectiv. Şabloanele de funcţii au avantajul unei determinări rapide a modurilor în care este folosit şablonul. Dacă programul nu specifică prototipuri de funcţii pentru un anumit şablon, compilatorul va putea să determine ce funcţii să creeze, pornind de la modul de folosire a funcţiei în program.

Următorul program, MIN_MAX.CPP, creează şi foloseşte şabloane pentru funcţiile *max_value* şi *min_value*.

```
#include <iostream.h>

template<class T> T max_value(T *array, int num_elements)
 {
   T max = array[0];

   for (int i = 0; i < num_elements; i++)
```

```
      if (max < array[i])
        max = array[i];

    return(max);
  }

template<class T> T min_value(T *array, int num_elements)
  {
    T min = array[0];

    for (int i = 0; i < num_elements; i++)
      if (min > array[i])
        min = array[i];

    return(min);
  }

float max_value(float *, int);

int max_value(int *, int);

float min_value(float *, int);

int min_value(int *, int);

void main(void)
  {
    int values[] = {5, 1, 6, 12, 7};

    float prices[] = { 1.1, 3.3, 4.4, 2.2, 5.5 };

    cout << "Max of integer values is " << max_value(values, 5) <<
            endl;
    cout << "Max of floating-point values is " <<
            max_value(prices, 5) << endl;

    cout << "Min of integer values is " << min_value(values, 5) <<
            endl;
    cout << "Min of floating-point values is " <<
            min_value(prices, 5) << endl;
  }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> MIN_MAX  <ENTER>
Max of integer values is 12
Max of floating-point values is 5.5
Min of integer values is 1
Min of floating-point values is 1.1
```




CLARIFICAREA DEFINIŢIEI UNUI ŞABLON

Aşa cum am văzut, cuvântul-cheie *template* permite definirea unor tipare pentru definiţii de funcţii. Când întâlnim pentru prima dată şabloane, declaraţia acestora poate fi puţin derutantă. De exemplu, următoarea instrucţiune începe cu o definiţie de şablon pentru funcţia *max_value:*

```
template<class T> T max_value(T *array, int num_elements)
```

Pentru o mai bună claritate, se pot folosi două linii pentru a prezenta antetul şablonului, după cum urmează:

```
template<class T>
T max_value(T *array, int num_elements)
  {
    T min = array[0];

    for (int i = 0; i < num_elements; i++)
      if (min > array[i])
        min = array[i];

    return(min);
  }
```

După cum se observă, o dată ce se trece de primul rând al definiţiei şablonului, aceasta seamănă cu o definiţie de funcţie standard.

ŞABLOANE CARE FOLOSESC MAI MULTE TIPURI

Aţi învăţat până acum că, atunci când C++ întâlneşte un prototip de funcţie pentru un şablon sau un apel de funcţie definită printr-un şablon, el creează o funcţie, substituind tipul specificat cu primul parametru al funcţiei. Fiecare din şabloanele examinate anterior a folosit un singur simbol de tip. În general, funcţiile lucrează cu diferite tipuri de parametri. Acest lucru este valabil şi pentru funcţiile definite prin şabloane. Să considerăm, de exemplu, că programul trebuie să compare suma valorilor unui tablou cu o anumită valoare limită. Dacă tabloul conţine numere întregi, funcţia *sum_and_compare* ar putea avea următoarea formă:

```
int sum_and_compare(int *array, long value, int num_elements)
  {
    long sum = 0;
```

```
  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return((sum > value) ? 1: 0);
}
```

Dacă tabloul ar conţine valori reale, funcţia ar putea arăta astfel:

```
int sum_and_compare(float *array, float value, int num_elements)
{
  float sum = 0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return((sum > value) ? 1: 0);
}
```

După cum se observă, funcţiile folosesc tipuri diferite de parametri pentru tablou, ca şi pentru variabila locală *sum*.

Pentru a crea un şablon pentru funcţia *sum_and_compare*, acesta trebuie să admită două tipuri, cum se indică mai jos:

```
template<class T1, class T2>
int sum_and_compare(T1 *array, T2 value, int num_elements)
{
  T2 sum = 0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return((sum > value) ? 1: 0);
}
```

În acest caz, C++ va folosi primii doi parametri specificaţi în prototipul funcţiei sau în apelul de funcţie pentru a determina tipul şablonului. În situaţia când programul ar folosi un prototip de funcţii, compilatorul C++ va substitui tipurile după cum se arată în continuare:

```
int sum_and_compare(int *array, long value, int num_elements);

   int sum_and_compare(T1 *array, T2 value, int num_elements)
   {
     T2 sum = 0;

     for (int i = 0; i < num_elements; i++)
       sum += array[i];

     return((sum > value) ? 1: 0);
   }

     int sum_and_compare(int *array, long value, int num_elements)
     {
       long sum = 0;

       for (int i = 0; i < num_elements; i++)
         sum += array[i];

       return((sum > value) ? 1: 0);
     }
```


După cum se observă, compilatorul foloseşte tipul primului parametru pentru simbolul *T1* şi tipul celui de al doilea parametru pentru simbolul *T2*. Următorul program USE_MULT.CPP, foloseşte un şablon pentru a crea funcţii ce admit tablouri de tip *int* sau *float*:

```
#include <iostream.h>

template<class T1, class T2>
int sum_and_compare(T1 *array, T2 value, int num_elements)
{
  T2 sum = 0;

  for (int i = 0; i < num_elements; i++)
    sum += array[i];

  return((sum > value) ? 1: 0);
}

int sum_and_compare(int *array, long value, int num_elements);

int sum_and_compare(float *array, float value, int num_elements);

void main(void)
{
  int values[] = { 1000, 20000, 30000, 4000, 500 };
  float prices[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

  if (sum_and_compare(values, 32767L, 5))
    cout << "Values will overflow an int value" << endl;

  if (sum_and_compare(prices, 25.5, 5))
    cout << "Values exceed 25.5" << endl;
  else
    cout << "Values are less than 25.5" << endl;
}
```



ŞABLOANE CARE ADMIT MAI MULTE TIPURI

La crearea şabloanelor de funcţii, apar situaţii când trebuie folosite două sau mai multe tipuri distincte. În acest caz, definiţia şablonului trebuie să specifice simbolurile pentru fiecare tip, precedate de cuvântul-cheie *template*, cum se arată mai jos:

```
template <class T1, class T2>
```

În acest caz, C++ va atribui simbolurilor *T1* şi *T2* tipurile primilor doi parametri ai funcţiei, substituind ulterior tipurile în corpul funcţiei, acolo unde este necesar.

UNDE SE PLASEAZĂ ŞABLOANELE

Şabloanele folosite într-un program se plasează într-un fişier antet care se include, ulterior, în program. În acest mod se reduce numărul de instrucţiuni din fişierul sursă al programului, facilitând citirea şi înţelegerea programului. Dacă un programator doreşte să examineze şabloanele, el poate deschide fişierul antet cu şabloane. În program, se recomandă folosirea prototipurilor de funcţii, pentru a ajuta programatorii care citesc programul să înţeleagă mai bine folosirea şabloanelor.

CREAREA ŞABLOANELOR DE CLASE

Aşa cum sunt funcţii care diferă numai prin tip, la fel există clase cu aceleaşi proprietăţi. De exemplu, în Capitolul 5, aţi creat o clasă *stack* care permite memorarea şi recuperarea unor valori de tip *int*:

```
class stack {
public:
  stack(int size);
  int push(int value);
  int pop(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  int *storage;    // Stack data buffer
  int elements;    // Number of values in the stack
  int isempty;     // True when stack is empty
  int isfull;      // True when stack is full
  int stack_size;  // Number of values the stack can store
};
```


Să presupunem acum că cerinţele programului s-au schimbat şi trebuie să memorăm valori de tip *int* şi *float*. În loc de a crea două clase diferite, se poate proiecta un şablon de clasă, din care apoi se poate crea un obiect stivă de tipul *int* şi un alt obiect stivă de tipul *float*.

Pentru a crea un şablon de clasă, se începe cu cuvântul-cheie *template*, urmat de simbolurile de clasă închise între paranteze unghiulare. În cadrul definiţiei clasei, simbolurile de tip clasă se vor folosi ca în exemplul următor:

```
template<class T>  ———— Cuvânt cheie template şi simbol de tip
class stack {
public:
  stack(int size);                 Utilizarea simbolului de tip
  T push(T value);
  T pop(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  T *storage;     // Stack data buffer
  int elements;   // Number of values in the stack
  int isempty;    // True when stack is empty
  int isfull;     // True when stack is full
  int stack_size; // Number of values the stack can store
};
```

În cazul în care funcţiile membru sunt definite în afara clasei, ele trebuie precedate de cuvântul-cheie *template* şi simbolurile de tip. De exemplu, funcţiile membru *push* şi *pop* devin:

```
template<class T>
T stack<T>::pop(void)
 {
   if (is_empty())
     return(0);
   else
     {
       if (--elements == 0)
         isempty = 1;

       isfull = 0;

       return(storage[elements]);
     }
 }

template<class T>
T stack<T>::push(T value)
 {
   if (is_full())
     return(0);
```

```
  else
    {
      isempty = 0;
      storage[elements++] = value;

      if (elements == stack_size)
        isfull = 1;

      return(value);
    }
}
```

Pentru a crea un obiect *stack* în program, se va folosi numele clasei şi al tipului în funcţia *main*, ca mai jos:

```
void main(void)
{
   stack<int> values(10);
   stack<float> prices(10);

   // Other statements here

}
```

Nume clasă stack
Tip obiect
Dimensiune dorită a stivei

Următorul program, STACKEXP.CPP, creează şi foloseşte două obiecte de tip *stack*. Unul dintre obiecte conţine valori de tip *int*, iar celălalt valori de tip *float*:

```
#include <iostream.h>
#include <iomanip.h>

template<class T>
class stack {
public:
  stack(int size);
  T push(T value);
  T pop(void);
  int is_empty(void) { return(isempty); };
  int is_full(void) { return(isfull); };
private:
  T *storage;            // Stack data buffer
  int elements;          // Number of values in the stack
  int isempty;           // True when stack is empty
  int isfull;            // True when stack is full
  int stack_size;        // Number of values the stack can store
};

template<class T>
```



```
stack<T>::stack(int size)
{
  storage = new T[size];   // Allocate memory for the stack
  elements = 0;
  isempty = 1;
  isfull = 0;
  stack_size = size;
}

template<class T>
T stack<T>::pop(void)
{
  if (is_empty())
    return(0);
  else
    {
      if (--elements == 0)
        isempty = 1;

      isfull = 0;

      return(storage[elements]);
    }
}

template<class T>
T stack<T>::push(T value)
{

  if (is_full())
    return(0);
  else
    {
      isempty = 0;
      storage[elements++] = value;

      if (elements == stack_size)
        isfull = 1;

      return(value);
    }
}

void main(void)
{
  stack<int> values(10);
  stack<float> prices(15);

  for (int i = 0; ! values.is_full(); i++)
```

```
  values.push(i);

 while (! values.is_empty())
   cout << values.pop() << endl;

 for (i = 0; ! prices.is_full(); i++)
   prices.push(i * 100.0);

 while (! prices.is_empty())
   cout << setiosflags(ios::showpoint) << prices.pop() << endl;
}
```

În exemplul anterior, programul plasează 10 valori în stiva de întregi şi 15 valori în stiva de valori reale. La compilarea şi execuţia programului, pe ecran vor fi afişate valorile din fiecare stivă în parte.

ANALIZA UNUI ALT EXEMPLU

Următorul program, LINKLIST.CPP creează o clasă *LinkList* ai cărei membri arată astfel:

```
struct Node {
 int value;
 Node *next;
 Node *previous;
};

class LinkList {
 public:
   LinkList(void);
   void show_list(void);
   Node *append_value(int);
 private:
   Node *first;
   Node *end;
};
```

După cum se observă, clasa ţine evidenţa unei liste cu legături duble, între primul şi ultimul său nod. Clasa foloseşte funcţia *show_list* pentru a afişa conţinutul listei, şi funcţia *append_value* pentru a insera valori la sfârşitul listei.

Următoarele instrucţiuni implementează întregul program LINKLIST.CPP:

```
#include <iostream.h>

struct Node {
 int value;
 Node *next;
 Node *previous;
```



```
};

class LinkList {
 public:
   LinkList(void);
   void show_list(void);
   Node *append_value(int);
 private:
   Node *first;
   Node *end;
};

LinkList::LinkList(void)
 {
   first = NULL;
   end = NULL;
 }

void LinkList::show_list(void)
 {
   Node *node;

   node = first;

   while (node)
     {
       cout << node->value << endl;
       node = node->next;
     }
 }

Node *LinkList::append_value(int value)
 {
   Node *ptr = end;

   end = new Node;

   if (first == NULL)
     first = end;
   else
     ptr->next = end;

   if (end)
     {
       end->next = NULL;
       end->previous = ptr;
       end->value = value;
     }
```

```
    return(end);
  }

void main(void)
  {
    LinkList list;

    for (int i = 0; i < 10; i++)
      list.append_value(i);

    list.show_list();

    for (i = 10; i < 20; i++)
      list.append_value(i);

    list.show_list();
  }
```

În exemplul precedent, lista cu legături lucrează numai cu elemente de tip *int*. Folosind un şablon de clasă, puteţi modifica rapid programul pentru a lucra cu valori de tip *int* şi *float*. Următorul program, LINKTEMP.CPP, foloseşte un şablon pentru a crea două astfel de liste:

```
#include <iostream.h>
#include <iomanip.h>

template<class T>
struct Node {
 T value;
 Node *next;
 Node *previous;
};

template<class T1>
class LinkList {
 public:
   LinkList(void);
   void show_list(void);
   Node<T1> *append_value(T1);
 private:
   Node<T1> *first;
   Node<T1> *end;
};

template<class T1> LinkList<T1>::LinkList(void)
 {
   first = NULL;
   end = NULL;
 }
template<class T1> void LinkList<T1>::show_list(void)
{
  Node<T1> *node;

  node = first;

  while (node)
    {
      cout << node->value << endl;
      node = node->next;
    }
}

template<class T1>
Node<T1> *LinkList<T1>::append_value(T1 value)
 {
   Node<T1> *ptr = end;

   end = new Node<T1>;

   if (first == NULL)
     first = end;
   else
     ptr->next = end;

   if (end)
     {
       end->next = NULL;
       end->previous = ptr;
       end->value = value;
     }

   return(end);
 }

void main(void)
 {
   LinkList<int> list;
   LinkList<float> values;

   for (int i = 0; i < 10; i++)
     list.append_value(i);

   list.show_list();

   cout << setiosflags(ios::showpoint);

   for (i = 1; i < 20; i++)
     values.append_value(i * 100.0);

   values.show_list();
 }
```

Lista cu legături este alcătuită din noduri, fiecare nod conţinând o valoare şi pointeri la nodul precedent şi nodul următor din listă, cum se arată în Figura 6.1.

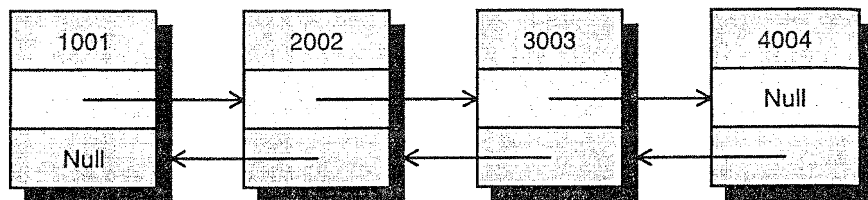


Figura 6.1. *Structura de noduri în cadrul listei cu legături*

Pentru a permite structurii de noduri să conţină valori de diferite tipuri, cum sunt *int* şi *float*, programul ar trebui să folosească un şablon, cum este următorul:

```
template<class T>
struct Node {
 T value;
 Node *next;
 Node *previous;
};
```

Apoi, programul foloseşte un şablon de clasă ce defineşte operaţiile pe o listă cu legături. Folosind acest şablon, clasa va putea fi folosită pentru a crea obiecte de tip listă cu legături ce conţin valori întregi, reale ş.a.m.d.

```
template<class T1>
class LinkList {
 public:
   LinkList(void);
   void show_list(void);
   Node<T1> *append_value(T1);
 private:
   Node<T1> *first;
   Node<T1> *end;
};
```

Programul va putea construi două obiecte de tip listă cu legături, una ce conţine valori întregi şi alta ce conţine valori reale, folosind următoarele instrucţiuni:

```
LinkList<int> list;

LinkList<float> values;
```

Când compilatorul C++ va întâlni definiţiile obiectelor, va substitui mai întâi simbolul de tip *T1* cu tipul actual. În cadrul clasei, compilatorul va crea structuri

de noduri, înlocuind simbolul *T* cu tipul care a substituit simbolul *T1*. În cazul unei liste de întregi, de exemplu, substituţiile vor fi:

```
LinkList<int> list;

   template<class T1>
   class LinkList {
    public:
      LinkList(void);
      void show_list(void);
      Node<T1> *append_value(T1);
    private:
      Node<T1> *first;
      Node<T1> *end;
   };

      class LinkList {
       public:
         LinkList(void);
         void show_list(void);
         Node *append_value(int);
       private:
         Node<int> *first;
         Node<int> *end;
      };

         template<class T>
         struct Node {
          T value;
          Node *next;
          Node *previous;
         };

            struct Node {
             int value;
             Node *next;
             Node *previous;
            };
```

Aşa cum se observă, compilatorul abordează asemenea şabloane într-o manieră pas cu pas. Când folosiţi şabloane complexe în programe, încercaţi să realizaţi substituţii similare pe instrucţiunile respective, pentru a înţelege mai bine folosirea şabloanelor.

REZUMAT

Şabloanele C++ permit compilatorului să creeze în mod automat funcţii şi clase ale căror parametri sau membri diferă numai prin tip. Folosirea compila-

torului pentru a crea copii de funcţii şi clase despre care se ştie că funcţionează corect micşorează cantitatea de cod care trebuie scrisă şi şansele de introducere a erorilor. Acest capitol a analizat şabloanele în detaliu. Înainte de trece la cu Capitolul 7, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Un şablon este o schiţă de funcţie sau clasă pe care compilatorul C++ o va concretiza cu tipurile de parametri furnizate. Şabloanele sunt ideale pentru funcţii sau clase apropiate între ele, ai căror parametri sau membri diferă numai prin tip.
- ✓ C++ acceptă şabloanele de funcţii şi şabloanele de clase.
- ✓ Pentru a crea un şablon de funcţie, se foloseşte cuvântul-cheie *template* şi simbolurile de tip între paranteze unghiulare, ca de exemplu *template* .<*class T1*>. Restul declaraţiei de funcţie rămâne neschimbat, cu excepţia faptului că se foloseşte simbolul T1 în locul unui nume de tip ca *int* sau *float*.
- ✓ Pentru ca ulterior să creaţi o funcţie cu parametri de un anumit tip, includeţi un prototip de funcţie, sau folosiţi funcţia pur şi simplu. În timpul compilării, C++ va determina corect tipurile pe care trebuie să le accepte, prin analiza tipurilor de parametri transmise funcţiei.
- ✓ Dacă şablonul de funcţii acceptă tipuri distincte de parametri, compilatorul va atribui tipurile aşa cum este specificat în prototipul funcţiei, sau de la stânga la dreapta, în funcţie de parametrii transferaţi funcţiei.
- ✓ Pentru a crea un şablon de clasă, scrieţi în faţa definiţiei clasei cuvântul-cheie *template* urmat de simbolurile de clasă, după cum s-a arătat.
- ✓ Pentru a crea obiecte folosind şabloane de clasă, specificaţi numele clasei, urmat de tipul de obiect dorit scris între paranteze unghiulare, ca de exemplu *ClassName <int> ObjectName*.
- ✓ Dacă şablonul de clasă acceptă tipuri distincte de parametri, compilatorul va atribui tipurile aşa cum este specificat în declaraţia obiectului, de la stânga spre dreapta. De exemplu, următoarea declaraţie foloseşte tipul *int* pentru primul simbol de tip şi tipul *float* pentru al doilea simbol: *ClassName <int, float> ObjectName*.
- ✓ Dacă un şablon de clasă foloseşte al doilea şablon, compilatorul va efectua substituţia tipurilor unul câte unul, începând cu primul tip de clasă menţionat.

CAPITOLUL 7
ACOMODAREA CU ALOCAREA MEMORIEI

Când programele folosesc mari cantităţi de informaţii, ca de exemplu rezultatele unor teste la o clasă de 50 de elevi sau fişele individuale a 100 de angajaţi, sau chiar un şir de caractere foarte lung, ele pot memora aceste informaţii fie în tablouri, fie într-o zonă de memorie alocată dinamic. Avantajele alocării dinamice a memoriei constau în faptul că necesarul de memorie corespunde mai exact necesităţilor programului, ca ulterior, după terminarea prelucrării, aceasta să fie eliberată şi disponibilizată în alte scopuri.

În limbajul C, funcţiile *alloc*, *calloc* şi *malloc* din biblioteca de funcţii permit alocarea dinamică a memoriei în timpul execuţiei. Când memoria nu mai este necesară, programele C o eliberează cu ajutorul funcţiei *free*. Deşi programele C++ pot realiza alocarea dinamică a memoriei folosind aceste funcţii de bibliotecă, se preferă folosirea operatorilor *new* şi *delete*. Capitolul de faţă examinează aceşti operatori în detaliu. Veţi învăţa nu numai cum se alocă memorie pentru variabile de tip *int*, *float* sau structuri, ci şi modul de alocare a memoriei pentru clasele C++. La sfârşitul acestui capitol, veţi înţelege următoarele:

- Cum se alocă memorie în programele C++
- Ce se întâmplă când operatorul *new* nu reuşeşte să aloce memorie
- Cum se poate specifica o funcţie care se execută automat când o cerere de memorie nu poate fi satisfăcută
- Cum se eliberează memoria anterior alocată
- Cum se iniţializează o locaţie de memorie cu o anumită valoare
- Cum se alocă memorie pentru un tablou de valori
- Cum se suprapun operatorii *new* şi *delete*

UN EXEMPLU SIMPLU DE ALOCARE A MEMORIEI

Următorul program, SORT_TEN.CPP, defineşte un tablou ce poate memora 10 valori întregi. Programul completează apoi tabloul cu 10 valori aleatoare, transferând apoi tabloul funcţiei *bubble_sort* pentru sortare. În final se afişează valorile sortate:

```
#include <iostream.h>
#include <stdlib.h>

void bubble_sort(int *array, int num_elements)
{
 for (int i = 0; i < num_elements; i++)
   for (int j = 0; j < num_elements; j++)
     if (array[i] < array[j])
       {
         int temp = array[i];
         array[i] = array[j];
         array[j] = temp;
       }
}

void main(void)
{
  int array[10];

  for (int i = 0; i < 10; i++)
    array[i] = random(SIZE);              // Values from 0 through SIZE

  bubble_sort(array, 10);

  for (i = 0; i < 10; i++)
    cout << array[i] << endl;
}
```

Observaţie: *Dacă compilatorul pe care îl folosiţi nu acceptă funcţia* ***random()****, atunci funcţia* ***rand()*** *ar trebui să fie disponibilă. Cu toate că nu sunt identice, cele două funcţii realizează acţiuni similare.*

La compilarea şi execuţia acestui program, pe ecran vor apărea 10 valori sortate. Când numărul de valori cu care programul lucrează este fix, de exemplu 10 valori întregi, se pot folosi tablouri similare celui din programul precedent. Dar dacă tabloul este folosit numai la începutul sau la sfârşitul programului, atunci se consumă inutil memorie pentru o durată îndelungată de timp. În astfel de situaţii, memoria pentru tablouri trebuie alocată dinamic, după cerinţele impuse.

Alocarea dinamică a memoriei reduce, de asemenea, numărul schimbărilor efectuate asupra programului, în cazul în care specificaţiile acestuia se modifică. Să presupunem, de exemplu, că programul precedent trebuie să sorteze şi să afişeze 20 de valori întregi. Pentru a schimba programul, trebuie înlocuită fiecare apariţie a numărului 10 cu 20 şi apoi trebuie recompilat programul.

O modalitate de a reduce numărul acestor schimbări este de a defini tabloul şi ciclurile în funcţie de o constantă, aşa cum se arată în programul următor, SORT_CON.CPP:


```
#define SIZE 20

void main(void)
{
  int array[SIZE];

  for (int i = 0; i < SIZE; i++)
    array[i] = random(SIZE);         // Values from 0 through SIZE

  bubble_sort(array, SIZE);

  for (i = 0; i < SIZE; i++)
    cout << array[i] << endl;
}
```

Observaţie: *Acest program nu conţine codul pentru funcţia* bubble_sort *descrisă anterior.*

În cazul când tabloul îşi va schimba dimensiunea, singura instrucţiune care trebuie modificată este definiţia dimensiunii acestuia:

```
#define SIZE 20
```

În multe cazuri, nu se pot cunoaşte cerinţele de memorie înainte de execuţia programului. Să presupunem că utilizatorul va furniza numărul de elemente ale tabloului. Dacă se folosesc tablouri, programul va trebui să aloce un tablou de dimensiune maximă şi să folosească numai elementele necesare. Următorul program, BIGARRAY.CPP, realizează acest lucru:

```
#define SIZE 30000

void main(void)
{
  int array[SIZE];
  int size;

  cout << "Type in the number of values to sort (less than " <<
          SIZE << "): ";
  cin >> size;

  if (size <= SIZE)
   {
     for (int i = 0; i < size; i++)
       array[i] = random(SIZE);     // Values from 0 through SIZE

     bubble_sort(array, size);

     for (i = 0; i < size; i++)
       cout << array[i] << endl;
   }
```

```
  else
    cerr << "Array size must be less than " << SIZE << endl;
}
```

***Observaţie:** Acest program nu conţine codul funcţiei* bubble_sort *descrisă anterior.*

În acest exemplu, programul declară un tablou de 30 000 de valori. Dacă utilizatorul are nevoie de mai puţine valori, o parte din tablou nu este folosită. Aşa cum vom vedea, o soluţie mai bună este de a aloca memoria dinamic.

***Observaţie:** Programul anterior a folosit valoarea 30 000 pentru dimensiunea tabloului. Această dimensiune nu a fost aleasă la întâmplare. Multe compilatoare pentru PC încă impun restricţia ca tabloul să nu deşească 64K (deoarece tabloul este memorat într-un segment de 64 K). Cum un element întreg ocupă 2 octeţi, atunci întregul tablou va conţine 30 000 × 2 = 60K, ceea ce se încadrează într-un segment, lăsând liber un mic spaţiu pentru alte utilizări.*

Următorul program, C_MALLOC.CPP, alocă memoria pentru tablou în mod dinamic, folosind funcţia de bibliotecă *malloc*. În acest mod programul poate aloca un tablou a cărui dimensiune este în concordanţă cu cerinţele utilizatorului:

```
#include <malloc.h>

#define SIZE 30000

void main(void)
 {
   int *array;
   int size;

   cout << "Type in the number of values to sort (less than " <<
        SIZE << "): ";
   cin >> size;

   if (size <= SIZE)
    {
     if ((array = (int *) malloc(size * sizeof(int))) == NULL)
       cerr << "Error allocating memory" << endl;
     else
      {
       for (int i = 0; i < size; i++)
         array[i] = random(SIZE);   // Values from 0 through SIZE

       bubble_sort(array, size);

       for (i = 0; i < size; i++)
         cout << array[i] << endl;
```



```
       free(array);
      }
    }
  else
    cerr << "Array size must be less than " << SIZE << endl;
}
```

***Observaţie:** Acest program nu conţine codul funcţiei* bubble_sort.

Prototipul pentru funcţia *malloc* se află în fişierul antet MALLOC.H. Funcţia *malloc* returnează un pointer la regiunea de memorie alocată, sau valoarea NULL dacă cererea de memorie nu este satisfăcută.

```
if ((array = (int *) malloc(size * sizeof(int))) == NULL)
    cout << "Error allocating memory" << endl;
```

După cum se observă, programul apelează *malloc* cu numărul de octeţi cerut, care în acest caz este numărul de elemente (*size*) înmulţit cu dimensiunea fiecărui element (*sizeof(int)*). După terminarea lucrului cu memoria alocată, programul o disponibilizează folosind funcţia de bibliotecă *free*. În mod prestabilit, funcţia *malloc* returnează un pointer de tip *void*. Simbolul (*int**) care precede funcţia converteşte explicit pointerul de tip *void* la un pointer de tip *int*.

Acest program restricţionează, de asemenea, dimensiunea tabloului, la mai puţin de 30 000. Ca şi în cazul tablourilor statice, multe compilatoare ce folosesc sistemul DOS impun ca memoria dinamică alocată de un program să nu depăşească un segment de 64 K. Veţi învăţa, puţin mai târziu, cum se determină cantitatea de memorie pe care programele o pot aloca dinamic.



LUCRUL CU MEMORIA DINAMICĂ

După cum se cunoaşte, tablourile permit programelor să memoreze informaţii de acelaşi tip. De exemplu, un tablou ar putea conţine rezultatele testelor a 50 de studenţi, altul ar putea conţine salariile a 100 de angajaţi. Deşi tablourile sunt foarte convenabile pentru gruparea datelor, ele pot introduce necesitatea efectuării multor schimbări în program când numărul de elemente conţinute trebuie să se modifice. În plus, dacă tablourile sunt folosite numai în anumite momente din cursul prelucrării datelor programului, ele consumă inutil memorie, chiar şi când nu sunt folosite. Ca o alternativă, programele pot aloca memoria dinamic, pe măsura cerinţelor. Memoria va fi astfel alocată în cantitatea necesară şi la momentul necesar. La terminarea lucrului cu memoria, programul o poate elibera, disponibilizând-o pentru alte necesităţi. Operatorul *new* din C++ realizează alocarea dinamică a memoriei, iar operatorul *delete* disponibilizează memoria alocată anterior.

ALOCAREA DINAMICĂ A MEMORIEI ÎN C++

Alocarea dinamică a memoriei în C++ se realizează cu ajutorul operatorilor *new* şi *delete*, care sunt asemănători cu funcţiile de bibliotecă *malloc* şi *free* discutate anterior. De exemplu, pentru a aloca dinamic un tablou de 100 de valori întregi, se foloseşte *new* astfel:

```
int *array = new int[100];
```

Ca şi în cazul funcţiei *malloc*, dacă operatorul *new* nu poate aloca memoria cerută, el va atribui pointerului respectiv valoarea NULL. În program, testarea atribuirii valorii NULL unui pointer se poate face astfel:

```
if (array == NULL)
  cerr << "Unable to allocate memory" << endl;
```

Observaţie: *Aşa cum veţi învăţa în continuare, unele compilatoare nu atribuie valoarea NULL când operatorul new nu poate aloca memoria cerută. În schimb, unele compilatoare stopează imediat programul (prin apariţia unei erori ce nu poate fi preluată, vezi Capitolul 14). Însă C++ vă permite să specificaţi o funcţie proprie, care intră în execuţie când memoria nu poate fi alocată. În acest fel, programele pot trata erorile de tip memorie insuficientă într-un mod personal.*

La terminarea lucrului cu memoria, aceasta va fi eliberată astfel:

```
delete array;
```

Următorul program, NEW_TEN.CPP, foloseşte operatorul *new* pentru a aloca un tablou de 10 valori întregi. Programul atribuie apoi tabloului valori aleatoare, după care îl sortează şi îi tipăreşte elementele. Când tabloul nu mai este necesar în program, operatorul *delete* va elibera zona de memorie corespunzătoare:

```
void main(void)
{
  int *array = new int[10];

  if (array == NULL)
    cerr << "Error allocating memory" << endl;
  else
    {
      for (int i = 0; i < 10; i++)
        array[i] = random(30000);   // Values from 0 thru 30.000

      bubble_sort(array, 10);

      for (i = 0; i < 10; i++)
        cout << array[i] << endl;

      delete array;
    }
}
```

Observaţie: *Acest program nu conţine şi codul funcţiei* bubble_sort.

După cum se observă, dacă operatorul *new* reuşeşte să aloce memoria, programul va trata tabloul alocat dinamic exact ca şi pe unul obişnuit. Următorul program, NEW_ASK.CPP, cere utilizatorului să introducă volumul de memorie necesar tabloului. Programul va aloca apoi tabloul folosind operatorul *new*:

```
#define SIZE 30000

void main(void)
{
  int *array;
  int size;

  cout << "Type in the number of values to sort (less than " <<
          SIZE << "): ";
  cin >> size;

  if (size <= SIZE)
   {
     if ((array = new int[size]) == NULL)
       cerr << "Error allocating memory" << endl;
     else
      {
        for (int i = 0; i < size; i++)
          array[i] = random(SIZE);   // Values from 0 thru SIZE

        bubble_sort(array, size);

        for (i = 0; i < size; i++)
          cout << array[i] << endl;

        delete array;
      }
   }
  else
    cerr << "Array size must be less than " << SIZE << endl;
}
```

Observaţie: *Acest program nu conţine şi codul funcţiei* bubble_sort.

Dacă memoria nu poate fi alocată, atunci valoarea NULL va fi atribuită pointerului.

Se poate observa că programul foloseşte operatorul *new* pentru a aloca memoria cerută într-o instrucţiune *if*:

```
if ((array = new int[size]) == NULL)
      cerr << "Error allocating memory" << endl;
```

CHEIA SUCCESULUI

ALOCAREA DINAMICĂ A MEMORIEI FOLOSIND OPERATORUL NEW

Operatorul *new* din C++ permite programului să aloce dinamic memoria. La folosirea operatorului *new* se specifică numărul de elemente cerute. Numărul total de octeţi va putea fi calculat pe baza tipului de date pe care programul îl alocă. Pentru un element de tip *int*, de exemplu, operatorul *new* alocă doi octeţi. Pentru tipul *float*, va aloca patru octeţi. Dacă operatorul *new* reuşeşte să aloce memoria, el va returna un pointer la zona de memorie alocată. Dacă operatorul *new* nu poate satisface cerinţa de memorie, el va returna valoarea NULL. Următoarea instrucţiune, de exemplu, foloseşte *new* pentru a aloca un tablou de 50 de elemente de tip *int*.

```
int *array = new int[50];
```

Tip element
Pointer la memoria alocată
Număr elemente
Tip element alocat

Analog, următoarea instrucţiune alocă un tablou de 100 elemente reale:

```
float *salaries = new float[100];
```

După ce programul alocă memorie cu operatorul *new*, trebuie testată valoarea pointerului pentru a ne asigura că operaţia a reuşit. Dacă *new* nu poate aloca memoria cerută, atunci el va atribui pointerului valoarea NULL:

```
if (salaries == NULL)
  cerr << "Error allocating memory" << endl;
```

Când în program nu mai este nevoie de memoria alocată, ea trebuie eliberată cu operatorul *delete*, pentru a fi ulterior refolosită:

```
delete array;
delete salary;
```

Programul anterior a folosit operatorul *new* pentru alocarea memoriei necesare unui tablou de tip *int*. În acelaşi mod se pot aloca zone de memorie pentru tablouri care să conţină informaţii de alt tip. Următorul program, SUMFLOAT.CPP, foloseşte operatorul *new* pentru a aloca un tablou de 15 valori

reale. Programul atribuie apoi valori aleatoare elementelor tabloului, după care afişează aceste valori şi suma lor:

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

#define SIZE 15

void main(void)
 {
   float *array = new float[SIZE];
   float sum = 0.0;

   if (array == NULL)
     cerr << "Error allocating memory" << endl;
   else
     {
       for (int i = 0; i < SIZE; i++)
         {
           array[i] = rand() / 1000.0;
           sum += array[i];
           cout << setw(6) << setprecision(2) <<
                   setiosflags(ios::fixed || ios::right) <<
                   array[i] << endl;
         }

       cout << "Array sum is " << sum << endl;
       delete array;
     }
 }
```

După cum se observă, programul alocă memorie cu operatorul *new* şi o eliberează cu operatorul *delete*.

Observaţie: *Dacă memoria alocată nu se eliberează cu operatorul delete, ea va fi totuşi automat eliberată la sfârşitul programului. Aveţi, totuşi, în vedere, că unul din motivele alocării dinamice este de a aloca memorie după necesităţi şi de a o elibera imediat ce nu mai este utilă, pentru a fi refolosită în alte scopuri.*

LUCRUL CU VARIABILE POINTER

Exemplele anterioare au alocat memorie pentru tablouri folosind operatorul *new*. Deoarece C++ tratează numele de tablou ca pe un pointer, nu trebuie să vă faceţi griji în privinţa variabilelor pointer şi a redirectării pointerilor. Dacă în program alocaţi memorie pentru altceva decât tablouri, atunci trebuie să folosiţi pointeri pentru accesul la memoria alocată. De exemplu, următorul program USE_PTRS.CPP, alocă memorie pentru valori de tip *int* şi *float*, atribuie valori

acestor locaţii de memorie şi afişează aceste valori referite prin intermediul pointerilor:

```
#include <iostream.h>

void main(void)
 {
   int *int_pointer;
   float *float_pointer;

   if ((int_pointer = new int) == NULL)
     cerr << "Error allocating memory for int" << endl;
   else
     {
       *int_pointer = 1001;
       cout << "The value assigned to memory location " <<
               int_pointer << " is " << *int_pointer << endl;
     }

   if ((float_pointer = new float) == NULL)
     cerr << "Error allocating memory for float" << endl;
   else
     {
       *float_pointer = 11.22;
       cout << "The value assigned to memory location " <<
               float_pointer << " is " << *float_pointer << endl;
     }
 }
```

După cum se observă, programul foloseşte operatorul de adresare indirectă când lucrează cu valori din locaţii de memorie referite prin pointeri. Observaţi instrucţiunea:

```
cout << "The value assigned to memory location " << float_pointer <<
        " is " << *float_pointer << endl;
```

Deoarece prima referinţă la *float_pointer* nu este precedată de asterisc, programul va afişa valoarea acestui pointer, care este o adresă de memorie. A doua referinţă la *float_pointer* este precedată de asterisc şi, prin urmare, programul va afişa valoarea conţinută în locaţia de memorie referită de *float_pointer*. La compilarea şi execuţia acestui program, pe ecran va apărea o situaţie similară celei de mai jos:

```
C:\> USE_PTRS  <ENTER>
The value assigned to memory location 0x14c0 is 1001
The value assigned to memory location 0x14c8 is 11.22
```

La compilarea şi rularea programului locaţiile de memorie pot să difere de la un sistem la altul, în funcţie de sistemul de operare folosit, de programele rezidente în memorie sau de programele de control ale dispozitivelor I/O. După cum se poate vedea, programul afişează valoarea pointerului (adresa) şi valoarea stocată în locaţia de memorie. Figura 7.1 arată cum pointerii adresează locaţii de memorie ce conţin, la rândul lor, valori de diferite tipuri.

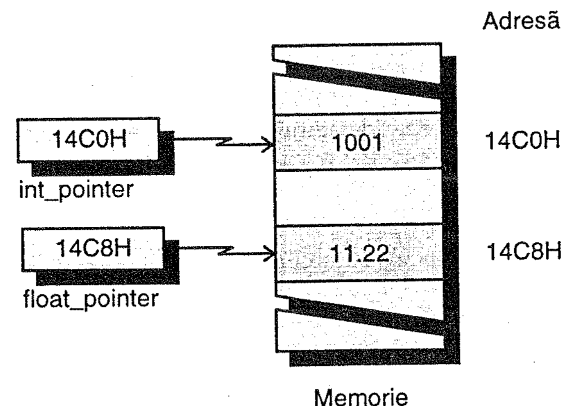


Figura 7.1. *Un pointer conţine adresa de memorie a unei valori sau adresa de început a unui domeniu de valori*

Să observăm că programul anterior a declarat variabilele pointer astfel:

```
int   *int_pointer;
float  *float_pointer;
```

Dacă veţi examina programe scrise în C sau C++, veţi întâlni şi declaraţii de pointeri care plasează asteriscul imediat după numele tipului acestora, ca mai jos:

```
int*   int_pointer;
float*  float_pointer;
```

De asemenea, puteţi întâlni declaraţii la care asteriscul se află între numele tipului de pointer şi numele variabilei:

```
int  *int_pointer;
float  *float_pointer;
```

Indiferent unde s-ar plasa asteriscul, toate cele trei forme creează variabile pointer identice. Pentru uniformitate, toate programele prezentate în această carte plasează asteriscul imediat înaintea numelui de variabilă.

INIŢIALIZAREA UNEI VALORI FOLOSIND OPERATORUL NEW

La alocarea dinamică a unor variabile simple, ca cele de tipul *int* sau *float*, poate fi necesară atribuirea imediată a unor valori care să fie stocate în acele

locaţii de memorie. De exemplu, următoarele instrucţiuni alocă un pointer de tip *int* apoi atribuie valoarea 1001 locaţiei de memorie respective:

```
int *int_ptr = new int;

if (int_ptr)
  *int_ptr = 1001;
```

Pentru mai multă comoditate, operatorul *new* permite specificarea unei valori iniţiale în momentul alocării, plasând valoarea dorită între paranteze, ca mai jos:

```
int *int_ptr = new int(1001);
```

Dacă *new* reuşeşte să aloce memorie, el va realiza şi iniţializarea. Dacă memoria nu poate fi alocată, iniţializarea este ignorată. Următorul program, NEW_INIT.CPP, foloseşte operatorul *new* pentru a aloca spaţiu de memorie şi apoi pentru a-l iniţializa cu valori de tip *int* sau *float*.

```
#include <iostream.h>

void main(void)
 {
   int *int_pointer = new int(1001);
   float *float_pointer = new float(3.12345);

   cout << *int_pointer << endl;
   cout << *float_pointer << endl;
 }
```

SĂ ÎNŢELEGEM ARITMETICA POINTERILOR

Mulţi programatori în C sau C++ folosesc pointeri în loc de operaţii cu tablouri, pentru îmbunătăţirea performanţei programelor. Să considerăm, de exemplu, următoarele funcţii, care afişează valorile într-un tablou de tip *float*:

```
void show_values(float array[], int size)
 {
   for (int i = 0; i < size; i++)
     cout << array[i] << endl;
 }
```

```
void show_values(float *array, int size)
 {
   for (int i = 0; i < size; i++)
     cout << *array++;
 }
```

Funcţia din stânga afişează valorile folosind un tablou şi operaţii de indexare. Funcţia din partea dreaptă, foloseşte un pointer pentru afişarea valorilor. Dacă includeţi cele două funcţii în programe diferite şi generaţi listingurile în limbaj de asamblare, veţi vedea că, în cazul folosirii tabloului, compilatorul va insera mai multe instrucţiuni decât în cazul pointerului. Drept urmare, implementarea cu pointeri conţine mai puţine instrucţiuni şi este mai rapidă.

Într-un program, un pointer indică o valoare de un anumit tip. Singura excepţie la această regulă sunt pointerii *void*, care nu au tip. Motivul pentru care pointerii

au tipuri asociate este de a cunoaşte cu câţi octeţi trebuie incrementat pointerul în memorie când se doreşte traversarea unei liste de valori ca cea anterioară. De exemplu, dacă folosiţi un pointer pentru a traversa un şir de caractere, compilatorul va incrementa conţinutul pointerului cu 1 la întâlnirea unei expresii de forma *string++*. Dacă folosiţi un pointer la un tablou de întregi, atunci compilatorul va incrementa pointerul cu doi octeţi pentru a adresa corect următorul element din tablou. Analog, pointerul către o listă de valori reale va fi incrementat cu 4. Dacă pointerii nu ar avea tipuri asociate, compilatorul nu ar putea realiza astfel de operaţii.

TRATAREA POINTERILOR CA TABLOURI

Multe programe C++ tratează tablourile drept pointeri. De exemplu, următorul program, SHOW_STR.CPP, transferă un tablou şir de caractere funcţiei *show_string* ce afişează, la rândul ei, câte un caracter din şir folosind un pointer:

```
#include <iostream.h>

void show_string(char *string)
 {
   while (*string)
     cout.put(*string++);
 }

void main(void)
 {
   char book[] = "Success with C++\n";

   show_string(book);
 }
```

Dacă examinaţi programe în C sau C++, veţi întâlni periodic coduri în care se combină operaţiile cu tablouri şi operaţiile cu pointeri. De exemplu, următorul fragment de program atribuie valoarea 3 celui de al patrulea element (de indice 3) dintr-un tablou de întregi:

```
void some_function(int *array, int size)
 {
   array[3] = 3;

   // Other statements
 }
```

Se observă că funcţia primeşte ca parametru un pointer la tipul *int*, după care tratează pointerul ca fiind adresa de început a unui tablou. Este important de reţinut că orice pointer, în orice moment, poate fi tratat ca baza unui tablou.

Următorul program, PTRARRAY.CPP, realizează exact opusul operaţiilor din programul precedent. În acest caz, funcţia *show_string* primeşte un pointer la un şir şi tratează pointerul ca fiind baza unui tablou:

```
#include <iostream.h>

void show_string(char string[])
{
  for (int i = 0; string[i]; i++)
    cout.put(string[i]);
}

void main(void)
{
  char *book = "Hi, Hello, world!\n";

  show_string(book + 4);
}
```

Să observăm că programul transferă funcţiei adresa de început a şirului plus 4 (se omit caracterele "Hi,"). Dacă rulaţi acest program pe ecran va apărea:

```
C:\> PTRARRAY  <ENTER>
Hello, world!
```

După cum se poate vedea, orice pointer poate fi tratat în orice moment că fiind baza unui tablou.

FOLOSIREA POINTERILOR CU PARAMETRII DE FUNCŢII

Una din cele mai folosite operaţii cu pointeri este schimbarea parametrilor unei funcţii. În mod prestabilit, la apelarea unei funcţii, C++ face copii ale valorilor parametrilor, punând aceste copii în stivă. Deoarece funcţia primeşte o copie a valorii parametrului, ea nu poate schimba valoarea acestuia. Următorul program, NOCHANGE.CPP, transferă 3 parametri funcţiei *one_two_three*, care, la rândul ei, atribuie parametrilor valorile 1,2 şi 3. Deoarece funcţia nu are acces la parametrii efectivi, ci la nişte copii ale acestora, parametrii rămân neschimbaţi după apelul funcţiei:

```
#include <iostream.h>

void one_two_three(int a, int b, int c)
{
  cout << "Values passed to the function: " << a << ' ' << b <<
           ' ' << c << endl;
  a = 1;
  b = 2;
  c = 3;
  cout << "Values within the function: " << a << ' ' << b <<
           ' ' << c << endl;
}
```



```
void main(void)
{
  int one = 0, two = 0, three = 0;
  one_two_three(one, two, three);
  cout << "Values in main: " << one << ' ' << two << ' ' <<
          three << endl;
}
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> NOCHANGE  <ENTER>
Values passed to the function: 0 0 0
Values within the function: 1 2 3
Values in main: 0 0 0
```

După cum se observă, valorile parametrilor actuali nu sunt modificate de funcţie. Pentru a putea schimba valorile parametrilor, programul trebuie să folosească pointeri. Parametrii funcţiei *one_two_three* trebuie să fie pointeri, cum se arată mai jos:

```
void one_two_three(int *a, int *b, int *c)
{
  cout << "Values passed to the function: " << *a << ' ' <<
          *b << ' ' << *c << endl;
  *a = 1;
  *b = 2;
  *c = 3;
  cout << "Values within the function: " << *a << ' ' << *b <<
          ' ' << *c << endl;
}
```

Parametrii funcţiei precedente sunt pointeri la valori de tip *int*. La apelul funcţiei, trebuie transferată adresa fiecărei variabile folosind operatorul de adresare (&), cum se arată în continuare:

```
one_two_three(&one, &two, &three);
```

Transferând funcţiei adresa variabilelor, aceasta va lucra cu valorile efective ale parametrilor, în loc de copiile acestor valori. În acest fel, schimbările efectuate asupra parametrilor rămân în vigoare şi după apelul funcţiei. Următorul program, PAR_CHG.CPP, foloseşte pointeri pentru a schimba valorile parametrilor în funcţia *one_two_three*:

```
#include <iostream.h>

void one_two_three(int *a, int *b, int *c)
{
  cout << "Values passed to the function: " << *a << ' ' <<
          *b << ' ' << *c << endl;
  *a = 1;
```

```
   *b = 2;
   *c = 3;
   cout << "Values within the function: " << *a << ' ' << *b <<
           ' ' << *c << endl;
 }

void main(void)
 {
   int one = 0, two = 0, three = 0;

   one_two_three(&one, &two, &three);

   cout << "Values in main: " << one << ' ' << two << ' ' <<
           three << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> PAR_CHG <ENTER>
Values passed to the function: 0 0 0
Values within the function: 1 2 3
Values in main: 1 2 3
```

Se observă că valorile transferate funcţiei se schimbă după apelul funcţiei. În Capitolul 10 se vor examina referinţele C++, care oferă o a doua posibilitate de a schimba valorile parametrilor într-o funcţie.

FOLOSIREA POINTERILOR LA STRUCTURI ŞI CLASE

Aşa cum se folosesc pointeri la variabile de tip *int*, *float*, *char* sau la tablouri de astfel de variabile, se pot folosi şi pointeri la structuri sau clase. Se ştie că, atunci când se lucrează cu structuri, se foloseşte operatorul punct (.) pentru a face referinţă la membrii structurii. De exemplu, următoarele instrucţiuni atribuie data de 25 decembrie 1994 structurii de tip *Date*:

```
struct Date {
  int month;
  int day;
  int year;
} xmas;

xmas.month = 12;
xmas.day = 25;
xmas.year = 1994;
```

Când lucrăm în program cu pointeri la structuri sau la clase, accesul la membri se poate face în unul din următoarele două moduri. Mai întâi, se poate folosi operatorul de indirectare şi pointerul între paranteze. De exemplu, următoarea funcţie, *assign_xmas*, atribuie data unei structuri indicată prin variabila *date_ptr*:


```
void assign_xmas(struct Date *date_ptr)
 {
    (*date_ptr).month = 12;
    (*date_ptr).day = 25;
    (*date_ptr).year = 1994;
 }
```

Deoarece această sintaxă poate încurca programatorii începători şi deoarece programatorii uită sistematic parantezele, limbajele C şi C++ oferă o a doua alternativă de a avea acces la membrii unei structuri indicaţi prin pointer sub forma următoare:

```
void assign_xmas(struct Date *date_ptr)
 {
    date_ptr->month = 12;
    date_ptr->day = 25;
    date_ptr->year = 1994;
 }
```

În cazul în care se lucrează cu clasele, aceleaşi două modalităţi pot fi folosite pentru accesul la membrii clasei.

POINTERII NU SUNT REFERINŢE

În Capitolul 10 veţi învăţa că o *referinţă* C++ este un nume asociat, sau un al doilea nume pe care programele îl pot folosi în cazul unei variabile. În general, referinţele sunt destinate simplificării funcţiilor care schimbă valorile parametrilor, eliminând necesitatea operaţiilor cu pointeri. De exemplu, următorul program, SWAP_PTR.CPP, foloseşte pointeri pentru a modifica valorile variabilelor *a* şi *b*:

```
#include <iostream.h>

void swap_values(int *first, int *second)
 {
   int temp = *first;
   *first = *second;
   *second = temp;
 }

void main(void)
 {
   int a = 1;
   int b = 1001;

   swap_values(&a, &b);

   cout << "Values after swap a: " << a << " b: " << b << endl;
 }
```

Analog, următorul program, SWAP_REF.CPP, foloseşte referinţe pentru schimbarea valorilor parametrilor:

```
#include <iostream.h>

void swap_values(int& first, int& second)
 {
   int temp = first;
   first = second;
   second = temp;
 }

void main(void)
 {
   int a = 1;
   int b = 1001;

   int& a_alias = a;    // Declare the references
   int& b_alias = b;

   swap_values(a_alias, b_alias);

   cout << "Values after swap a: " << a << " b: " << b << endl;
 }
```

Se observă că referinţele elimină necesitatea folosirii operaţiilor cu pointeri într-o funcţie. Dar utilizarea referinţelor măreşte complexitatea programelor, prin introducerea unor variabile suplimentare. Capitolul 10 va examina referinţele în detaliu. Deocamdată, reţineţi că o referinţă nu este un pointer.

ATENŢIE LA POINTERI ŞI LA VARIABILE LOCALE

Pe măsură ce programele devin mai complicate, apar situaţii când memoria trebuie alocată din interiorul funcţiilor. Este important să reţinem că, atunci când memoria este alocată din interiorul unei funcţii, ea rămâne alocată şi după ce funcţia se încheie. Să considerăm, de exemplu, programul LOCALPTR.CPP, care alocă 1000 de octeţi în cadrul funcţiei *bad_pointer*.

```
#include <iostream.h>

void bad_pointer(void)
 {
   char *ptr = new char[1000];        // Allocate the memory

   // Perform some operation
 }

void main(void)
 {
   // Invoke function
   bad_pointer();

   // Perform some operation

   // Invoke the function a second time
   bad_pointer();
 }
```


La fiecare apel al funcţiei *bad_pointer*, aceasta alocă memorie, atribuind adresa zonei alocate variabilei locale *ptr*. Din păcate, funcţia nu eliberează memoria adresată atunci când nu mai este nevoie de aceasta. Deoarece memoria este adresată folosind o variabilă locală, programul nu are nici o posibilitate de a disponibiliza memoria după terminarea funcţiei. Prin urmare, memoria este alocată, dar programul nu o poate accesa. Dacă într-un program se alocă temporar memorie din interiorul unei funcţii, trebuie să ne asigurăm că el va elibera memoria înainte ca funcţia să se încheie. Există însă şi situaţii, cum se va vedea în continuare, când se doreşte ca memoria să rămână alocată şi după încheierea funcţiei.

FUNCŢII CARE RETURNEAZĂ POINTERI

În funcţie de destinaţia unei funcţii, există situaţii când memoria alocată într-o funcţie trebuie să rămână alocată şi după încheierea funcţiei. De exemplu, următorul program, STR_DUP.CPP, foloseşte funcţia *string_duplicate*, care alocă memorie pentru a reţine caracterele dintr-un parametru de tip şir de caractere, returnând apoi un pointer către începutul noului şir. Dacă funcţia nu reuşeşte să copieze şirul, funcţia va returna NULL.

```
#include <iostream.h>
#include <string.h>

char *string_duplicate(char *source)
 {
   char *target = new char[strlen(source)];

   if (target)
     for (int i = 0; target[i] = source[i]; i++)
       ;

   return(target);
 }

void main(void)
 {
   char *new_string;

   new_string = string_duplicate("Success with C++!");

   cout << new_string << endl;
 }
```

Deoarece programul trebuie să aibă acces la zona de memorie alocată de funcție și după încheierea acesteia, funcția nu eliberează memoria cu operatorul *delete*.

SPAȚIUL LIBER DE MEMORIE

Memoria este alocată dinamic dintr-o zonă rezervată de memorie, denumită *spațiul liber* sau *zonă fragmentată* (heap). Așa cum se arată în Figura 7.2, spațiul liber este situat deasupra datelor programului și sub stiva programului.

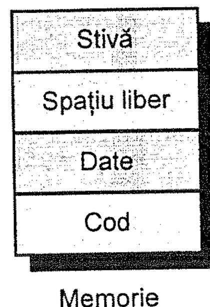


Fig. 7.2. Spațiul liber este situat între datele și stiva programului.

În funcție de sistemul de operare folosit și de modelul de memorie a compilatorului, cantitatea de spațiu liber disponibil va diferi de la un sistem la altul. Capitolul 15 va analiza în detaliu spațiul liber de memorie. Veți învăța ce se întâmplă la nivelul sistemului când se alocă sau se eliberează memorie. Deocamdată, însă, rețineți că programele nu au la dispoziție o cantitate nelimitată de memorie liberă. Este posibil ca unele programe să consume tot spațiul de memorie disponibilă.

CONDIȚIILE DE MEMORIE INSUFICIENTĂ

Așa cum ați învățat, când operatorul *new* nu poate aloca cantitatea de memorie necesară, el atribuie valoarea NULL variabilei pointer corespunzătoare. Următorul program, TAKE_ALL.CPP, folosește operatorul *new* într-un ciclu pentru a aloca blocuri a câte 10 K de memorie, până când nu mai rămâne memorie disponibilă. Programul afișează apoi un mesaj indicând cantitatea de memorie alocată în total:

```
#include <iostream.h>

void main(void)
```



```
char *buffer;
long sum = 0L;

do {
  buffer = new char[10000];
  if (buffer)
    {
      cout << "Allocated 10,000 bytes" << endl;
      sum += 10000;
    }
} while (buffer);

cout << "The amount of memory allocated was " << sum <<
        " bytes" << endl;
}
```

Compilați și executați acest program. În majoritatea cazurilor, operatorul *new* va atribui valoarea NULL pointerului *buffer*. Unele compilatoare, însă, vor stopa programul când memoria cerută nu poate fi alocată, programul afișând rezultate de forma:

```
C:\> TAKE_ALL  <ENTER>
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Abnormal program termination
```

Pentru a înțelege de ce programul se oprește brusc în acest mod, trebuie să știm că C++ permite specificarea unei funcții care urmează a fi executată când cererea de memorie nu poate fi satisfăcută. O astfel de funcție este *set_new_handler*, al cărui prototip se află definit în fișierul antet NEW.H.

Următorul program, SET_NEW.CPP, folosește *set_new_handler* pentru a apela funcția *message_and_exit* când cererea de memorie nu poate fi satisfăcută. Când funcția *message_and_exit* este apelată, se afișează un mesaj la fluxul *cerr* și programul este încheiat.

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void message_and_exit(void)
 {
   cerr << "\a\aNo more memory!" << endl;
   exit(1);
 }
```

```
void main(void)
 {
   char *buffer;
   long sum = 0L;

   set_new_handler(message_and_exit);

   do {
     buffer = new char[10000];
     if (buffer)
      {
        cout << "Allocated 10,000 bytes" << endl;
        sum += 10000;
      }
   } while (buffer);

   cout << "The amount of memory allocated was " << sum <<
         " bytes" << endl;
 }
```

Observaţie: *Numele exact şi sintaxa folosită de alte compilatoare pentru funcţia **set_new_handler** poate fi diferită de cea indicată aici.*

În marea majoritate a cazurilor de cerere nesatisfăcută de memorie, nu se doreşte lansarea în execuţie a unei funcţii, ci pur şi simplu returnarea rezultatului NULL de către operatorul new. În astfel de situaţii se poate invoca funcţia set_new_handler cu parametrul NULL pentru dezactivarea funcţiei. Următorul program, USE_NULL.CPP, foloseşte acest apel:

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void main(void)
 {
   char *buffer;
   long sum = 0L;

   set_new_handler(NULL);

   do {
     buffer = new char[10000];
     if (buffer)
      {
        cout << "Allocated 10,000 bytes" << endl;
        sum += 10000;
      }
   } while (buffer);

   cout << "The amount of memory allocated was " << sum <<
         " bytes" << endl;
 }
```


Aşa cum veţi vedea în secţiunea următoare, cea mai bună metodă de a controla operaţiile de alocare a memoriei este suprapunerea operatorilor *new* şi *delete*.



SPECIFICAREA UNEI FUNCŢII PENTRU TRATAREA ERORILOR DE MEMORIE INSUFICIENTĂ

În mod normal, dacă operatorul *new* nu poate satisface o cerere de memorie, se va atribui valoarea NULL pointerului corespunzător. În anumite situaţii (de exemplu, atunci când se testează programul) se poate dori afişarea unui mesaj de eroare la cerere de memorie nesatisfăcută. Terminând programul în acest mod, se poate localiza o operaţie de alocare a memoriei al cărei rezultat nu este verificat la NULL. Pentru a specifica o funcţie ce urmează să fie executată la cerere de memorie nesatisfăcută, se poate folosi funcţia *set_new_handler*. Fişierul antet NEW.H furnizează următorul prototip:

```
set_new_handler(* new_handler());
```

Pentru a invalida apelul unei funcţii speciale de tratare a erorilor, se poate realiza un apel cu parametrul NULL:

```
set_new_handler(NULL);
```

SUPRAPUNEREA OPERATORILOR NEW ŞI DELETE

În Capitolul 5 aţi învăţat cum se suprapun operatorii în program. Deoarece *new* şi *delete* sunt operatori C++, şi ei pot fi suprapuşi. Aşa cum veţi vedea în Capitolul 15, C++ nu realizează aşa-numita *colectare a resturilor* pentru gruparea spaţiilor fragmentate de memorie într-o zonă mai mare, în scopul satisfacerii cerinţelor de memorie. De exemplu, în Figura 7.3, spaţiul liber conţine memorie alocată şi disponibilă. Se observă că există 500 de octeţi de memorie liberă. Din păcate, cel mai mare bloc de memorie disponibilă are 250 de octeţi. Dacă programul trebuie să aloce 300 de octeţi, alocarea va eşua.

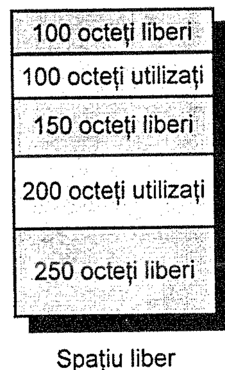


Figura 7.3. *Locațiile fragmentate de spațiu liber împiedică alocarea memoriei.*

Colectarea resturilor este o tehnică folosită pentru a grupa memoria disponibilă într-un bloc compact. Folosind tehnicile de colectare a resturilor, spațiul liber disponibil poate fi grupat ca în Figura 7.4.

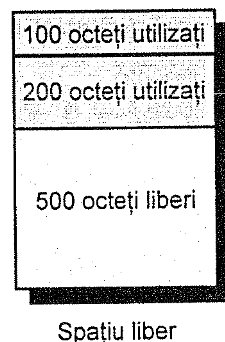


Figura 7.4. *Gruparea spațiului liber disponibil.*

Pentru a realiza colectarea resturilor, blocurile de memorie alocate și disponibile sunt deplasate pentru a crea un bloc mare de memorie disponibilă. C++ nu poate realiza colectarea de resturi, deoarece nu cunoaște cum sunt folosiți pointerii de către program în diferite regiuni de memorie. Totuși, programul poate realiza astfel de operații, deoarece are posibilitatea de a ține o evidență a folosirii pointerilor.

Un mod de a implementa colectarea resturilor este de a suprapune operatorii *new* și *delete* cu funcții care realizează periodic astfel de operații.

Din păcate, pașii ce trebuie realizați pentru a implementa o colectare de resturi eficientă și utilă depășește cadrul discuției din acest paragraf. Dar suprapunerea operatorilor *new* și *delete* nu este o chestiune complicată. Pentru aceasta, trebuie declarate funcții asemănătoare celor ce urmează:

```
void *operator new(size_t size)
 {
   // Statements
 }

void *operator new[](size_t size)
 {
   // Statements
 }

static void operator delete(void *pointer)
 {
   // Statements
 }
```

Se poate observa că majoritatea compilatoarelor furnizează două definiții pentru operatorul *new*, una care alocă memorie pentru tipuri simple de date (*int*, *float*), și alta care alocă memoria pentru tablouri. În mod normal, în cadrul operatorului *new* se alocă memorie cu funcția *malloc*, ca mai jos:

```
void *operator new(size_t size)
 {
   // Statements

   return(malloc(size));
 }
```

În interiorul operatorului *delete*, memoria se va elibera cu funcția *free*:

```
static void operator delete(void *pointer)
 {
   // Statements

   free(pointer);
 }
```

Suprapunerea operatorilor *new* și *delete* cere o oarecare experiență, în funcție de compilatorul folosit. De exemplu, următorul program, LIM_1000.CPP, suprapune operatorul *new* pentru a nu aloca decât blocuri ce nu depășesc 1000 de octeți. În cazul când programul apelează *new* cu o valoare mai mare decât 1000, acesta va returna imediat valoarea NULL:

```
#include <iostream.h>
#include <alloc.h>
```

```
void *operator new[](size_t size)
{
  if (size >= 1000)
    return(0);

  return(malloc(size));
}

void main(void)
{
  char *string = new char[1000];
  float *salaries = new float[200];
  int *ages = new int[250];

  if (string)
    cout << "string[1000] was allocated" << endl;
  else
    cout << "Error allocating string[1000]" << endl;

  if (salaries)
    cout << "salaries[200] was allocated" << endl;
  else
    cout << "Error allocating salaries[200]" << endl;

  if (ages)
    cout << "ages[250] was allocated" << endl;
  else
    cout << "Error allocating ages[250]" << endl;
}
```

Observaţie: *Alte compilatoare pot folosi fişierul antet* malloc.h *în loc de* alloc.h

În exemplul precedent, programul suprapune operatorul *new* [ ]. După cum se observă, programul încearcă să aloce memorie pentru diferite tipuri de tablouri, ale căror necesităţi de memorie sunt 1000, 8000 (4*200 octeţi) şi 500 (2*250 octeţi). La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> LIM_1000 <ENTER>
Error allocating string[1000]
salaries[200] was allocated
ages[250] was allocated
```

În mod normal, programele nu vor ignora operatorii globali *new* şi *delete*. În schimb, programele vor ignora operatorii pentru un anumit tip de clasă. Următorul program, OVER_NEW.CPP, suprapune operatorul *new* pentru a fi folosit cu clasa *Date* în scopul memorării în buffer-ul de date a valorii prestabilite 1 ianuarie 1994. O prelucrare similară s-ar putea realiza, de asemenea, pentru încărca data curentă în buffer:


```
#include <iostream.h>
#include <alloc.h>

class Date {
  public:
    void *operator new(size_t size);
    int month;
    int day;
    int year;
};

class DateClass {
  public:
    DateClass(int month, int day, int year);
    void show_date(void);
  private:
    Date *buffer;
};

DateClass::DateClass(int month, int day, int year)
{
  buffer = new Date;

  cout << "Default date: ";
  show_date();

  buffer->month = month;
  buffer->day = day;
  buffer->year = year;
}

void *Date::operator new(size_t size)
{
  Date *ptr = (Date *) malloc(size);

  if (ptr)
   {
     ptr->month = 1;
     ptr->day = 1;
     ptr->year = 1994;
   }

  return((void *) ptr);
}

void DateClass::show_date(void)
{
  cout << buffer->month << '/' << buffer->day << '/' <<
          buffer->year << endl;
}
```

```
void main(void)
{
  DateClass birthday(9, 30, 94);

  cout << "Birthday date: ";
  birthday.show_date();
}
```

Observaţie: *Alte compilatoare pot folosi fişierul antet* malloc.h *în loc de* alloc.h.

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> OVER_NEW  <ENTER>
Default date: 1/1/1994
Birthday date: 9/30/94
```

Se observă că, prin suprapunerea operatorilor *new* şi *delete*, programele pot câştiga enorm în controlul operaţiilor de alocare a memoriei.

REZUMAT

Aşa cum aţi învăţat, alocarea dinamică a memoriei are o serie de avantaje. În primul rând, programele pot determina mai precis necesarul de memorie în timpul execuţiei. În plus, alocarea dinamică a memoriei, în locul folosirii tablourilor, reduce numărul de schimbări efectuate în program în cazul modificării cerinţelor de memorie. În sfârşit, alocarea memoriei atunci când este nevoie şi eliberarea acesteia imediat ce nu mai este necesară permit ca memoria să fie folosită în alte scopuri. În Capitolul 8 veţi învăţa cum să folosiţi funcţii virtuale în cadrul programelor C++. Înainte de a trece la Capitolul 8, asiguraţi-vă că aţi învăţat următoarele:

- ✓ În limbajul C, pentru alocarea memoriei se folosesc funcţiile de bibliotecă *malloc* şi *calloc*, iar pentru eliberarea memoriei se foloseşte funcţia *free*. În C++, programele alocă memorie folosind operatorul *new* şi o eliberează ulterior folosind operatorul *delete*.
- ✓ Dacă operatorul *new* nu poate aloca memorie, atunci va atribui valoarea NULL pointerului corespunzător. Unele compilatoare pot însă să apeleze o funcţie care să încheie execuţia programului.
- ✓ Folosind funcţia *set_new_handler* se poate specifica o funcţie care se execută automat când o cerere de memorie nu poate fi satisfăcută. În funcţie de destinaţia programului, această funcţie ar putea încerca să elibereze memoria astfel încât cererea să fie repetată. Dacă se apelează funcţia *set_new_handler* cu parametrul NULL, atunci nu se va executa nici o funcţie de tratare a erorilor de memorie insuficientă.

- ✓ Când nu mai este nevoie de memoria alocată anterior, programele o pot elibera folosind operatorul *delete*. La terminarea unui program, toată memoria alocată anterior va fi eliberată automat.
- ✓ Pentru a aloca memorie pentru o variabilă simplă, ca *int* sau *float*, se foloseşte operatorul *new* astfel:

```
int *int_pointer = new int;
float *float_pointer = new float;
```

- ✓ Dacă în operatorul *new* se include între paranteze rotunde o valoare, atunci locaţia de memorie alocată este iniţializată cu această valoare (în cazul când cererea de memorie a fost satisfăcută):

```
int *int_pointer = new int(1001);
```

- ✓ Pentru a aloca memorie pentru un tablou de valori folosind operatorul *new*, se specifică numărul de elemente între paranteze pătrate, ca mai jos:

```
int *int_array = new int[50];
```

- ✓ În funcţie de scopul unui program, pot apărea situaţii când se doreşte ignorarea operatorilor *new* şi *delete*. În mod normal, programele nu vor ignora operatorii globali, ci mai degrabă operatorii folosiţi de o anumită clasă.

CAPITOLUL 8
ACOMODAREA CU FUNCŢIILE VIRTUALE ŞI POLIMORFISMUL

Este greu de găsit o carte sau un articol de revistă de C++ în care să nu se discute problema *polimorfismului*. Din păcate, în majoritatea cazurilor polimorfismul şi folosirea funcţiilor virtuale par un lucru dificil. Aşa cum veţi învăţa în acest capitol, partea cea mai dificilă a polimorfismului este termenul însuşi. O dată ce treceţi de acesta, conceptele sunt mai simple. Aşa că, să exemplificăm termenul chiar acum. *Poly*, în limba greacă, înseamnă multe, iar *morfic*, înseamnă forme. Când combinaţi cei doi termeni, obţineţi *multe forme*. Pe scurt, un obiect polimorfic este un obiect capabil de a avea două sau mai multe forme. De exemplu, să presupunem că avem un obiect de tip telefon. Pentru a realiza o convorbire, veţi folosi funcţia membru *dial* (formează_număr). Aşa cum se ştie, un telefon poate folosi fie o claviatură cu butoane, fie un disc rotativ. Cu alte cuvinte, telefoanele pot avea una din cele două *forme*. În funcţie de forma folosită, un obiect *telefon* va folosi o funcţie membru *dial* diferită pentru formarea numărului. Acest capitol explică ce este şi ce nu este polimorfismul. Veţi învăţa:

- Ce este polimorfismul şi la ce foloseşte
- Cum suportă C++ polimorfismul
- Ce este o funcţie virtuală şi cum se defineşte
- Cum se înlocuieşte o funcţie virtuală cu o funcţie proprie
- În ce condiţii are loc polimorfismul
- Ce este o funcţie pur virtuală

În Capitolul 13 veţi analiza mai în detaliu ce înseamnă polimorfismul pentru compilatorul C++.

SĂ ÎNŢELEGEM FUNCŢIILE VIRTUALE

C++ implementează polimorfismul folosind *funcţiile virtuale*. În sensul cel mai simplu, o funcţie virtuală este o funcţie membru proiectată pentru a funcţiona cu orice membru al clasei de bază sau al unei clase derivate (al cărui tip poate fi necunoscut). În acest fel, clasele pot alege să folosească funcţia virtuală sau o altă funcţie proprie, scutind astfel programatorii de a scrie diferite funcţii pentru diferite clase. O funcţie virtuală poate fi închipuită ca o "funcţie variabilă", adică o funcţie membru de clasă concepută pentru a fi înlocuită ulterior cu o funcţie membru necunoscută a unei clase derivate. Prin definirea unei funcţii drept *vir-*

tuală, un program poate crea obiecte cu propriile funcţii membru, folosite în locul funcţiei virtuale, cât şi obiecte care folosesc funcţia prestabilită.

Declaraţia unei funcţii virtuale este precedată de cuvântul-cheie *virtual*. De exemplu, următoarea clasă de bază *telephone* defineşte funcţia membru *dial* ca fiind *virtuală*, astfel încât să poată lucra cu telefoane de tip *rotary* (cu disc rotativ) şi *touch_tone* (cu taste):

```
class telephone {
  public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
  protected:
    char phone_number[32];
    int volume;
};
```

Cuvântul – cheie virtual

Următoarele clase derivate, *rotary* şi *touch_tone*, sunt bazate pe clasa *telephone*:

```
class rotary: public telephone {
  public:
    rotary(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};

class touch_tone: public telephone {
  public:
    touch_tone(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};
```

După cum se observă, ambele clase definesc funcţia membru *dial* (formează_număr). Următorul program, TELEPHON.CPP, foloseşte aceste două clase şi funcţiile *dial* aferente:

```
#include <iostream.h>
#include <string.h>

class telephone {
  public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
  protected:
    char phone_number[32];
    int volume;
};

telephone::telephone(char *number, int volume)
 {
```



```
   strcpy(telephone::phone_number, number);
   telephone::volume = volume;
 }

int telephone::dial(char *outgoing_number)
 {
   cout << "Use a rotary or touch-tone phone to call: " <<
         outgoing_number;

   cout << " Volume: " << volume << endl;
   return(1);
 }

class touch_tone: public telephone {
  public:
    touch_tone(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
 {
   cout << "Beep beep beep with touch tone: " << outgoing_number;
   cout << " Volume: " << volume << endl;

   return(1);
 }

class rotary: public telephone {
  public:
    rotary(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
 {
   cout << "Click click click with rotary: " << outgoing_number;
   cout << " Volume: " << volume << endl;

   return(1);
 }

void main(void)
 {
   touch_tone office("363-1111", 5);
   rotary home("555-1234", 2);

   home.dial("222-3333");
   office.dial("333-4444");
 }
```

Aşa cum se observă, programul declară obiectele telefon *rotary* şi *touch_tone*, apoi le foloseşte pentru a face o legătură în vederea unei convorbiri. La compilarea şi execuţia programului, pe ecran va apare:

```
C:\> TELEPHON  <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
```

În acest exemplu, programul foloseşte funcţii virtuale, dar nu foloseşte polimorfismul. Cu alte cuvinte, obiectele folosite în program nu au mai multe forme - fiecare obiect este fie telefon cu disc (*rotary*) fie telefon cu taste (*touch_tone*), dar nu de ambele tipuri.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM FUNCŢIILE VIRTUALE

Polimorfismul din C++ se bazează pe *funcţii virtuale*. În sensul cel mai simplu, o funcţie virtuală este o funcţie membru a unei clase care poate fi înlocuită, în momentul execuţiei programului, cu o funcţie membru derivată. Este un fel de funcţie "variabilă" pe care clasele derivate o pot substitui cu propriile lor funcţii membru.

Pentru a declara o funcţie virtuală, aceasta trebuie precedată de cuvântul-cheie *virtual*. De exemplu, clasa *computer* de mai jos foloseşte o funcţie membru *processor* care este virtuală:

```
class computer {
    public:
      computer(int processor_type);
      virtual processor(unsigned *instructions);
      void load_program(char *program_name);
   private:
      int processor_type;
      int memory_size;
};
```

În acest fel, programul ar putea deriva, de exemplu, clasele PC şi Mac, fiecare folosind o funcţie membru *processor* specifică.

Programul anterior a folosit funcţii virtuale, dar nu a realizat polimorfismul. Următorul program, POLY.CPP, modifică funcţia *main* pentru a folosi un pointer la un obiect de tip *telephone* denumit *cellular*. În timpul execuţiei programului pointerul face referinţă atât la un obiect de tip *rotary*, cât şi la unul de tip *touch_tone*:

```
void main(void)
 {
   touch_tone office("363-1111", 5);
   rotary home("555-1234", 2);

   telephone *cellular;

   home.dial("222-3333");
   office.dial("333-4444");

   cellular = &home;
   cellular->dial("555-1234");

   cellular = &office;
   cellular->dial("555-5678");
 }
```

Observaţie: *Acest program nu conţine şi definiţiile claselor şi funcţiilor membru. După cum se observă, acest program atribuie variabilei pointer* cellular *adresele a două tipuri diferite de obiecte. La compilarea şi execuţia acestui program, ecranul va afişa următoarele:*

```
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Click click click with rotary: 555-1234 Volume: 2
Beep beep beep with touch tone: 555-5678 Volume: 5
```

Când pointerului *cellular* i se atribuie tipuri de obiecte diferite, programul va apela funcţii distincte pentru funcţia membru *dial*. Cu alte cuvinte, acest program realizează polimorfism, obiectul *cellular* luând diferite forme.

Exersaţi cu acest program, folosind un obiect de tipul *telephone*, în locul unui pointer la obiect. Următorul program, NOPOLY.CPP, atribuie obiectului *cellular* cele două obiecte derivate, *rotary* şi *touch_tone*. Veţi vedea, totuşi, că obiectul nu îşi schimbă forma, rămânând în permanenţă un obiect de tip *telephone*:

```
void main(void)
 {
   touch_tone office("363-1111", 5);
   rotary home("555-1234", 2);

   telephone cellular("555-5555", 3);

   home.dial("222-3333");
   office.dial("333-4444");

   cellular = home;
   cellular.dial("555-1234");
    cellular = office;
   cellular.dial("555-5678");
 }
```

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> NOPOLY  <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Use a rotary or touch-tone phone to call: 555-1234 Volume: 2
Use a rotary or touch-tone phone to call: 555-5678 Volume: 5
```

După cum se observă, când programul atribuie obiectului *cellular* cele două obiecte *rotary* şi *touch_tone*, forma obiectului *cellular* nu se schimbă. Cu alte cuvinte, obiectul *cellular* continuă să folosească funcţia *dial* a clasei *telephone*.

Observaţie: *Acest program nu conţine şi definiţiile claselor şi funcţiile membru.*

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM POLIMORFISMUL

Polimorfismul este un alt mod de a spune „forme multiple". Un obiect polimorfic este un obiect capabil de a-şi schimba forma în timpul perioadei sale de existenţă. Polimorfismul este un instrument de programare puternic, deoarece permite scrierea unor fragmente de program ce se comportă corect pentru obiecte de tipuri diferite. În majoritatea cazurilor, tipul obiectului efectiv nu este cunoscut, nici chiar în timpul execuţiei. Aşa cum veţi învăţa în Capitolul 13, polimorfismul se poate manifesta datorită unor operaţii efectuate de compilator „în spatele cortinei". Deocamdată însă, să reţinem că funcţiile virtuale C++ ne permit să scriem programe ce suportă polimorfismul, programe în care un obiect poate trece de la o formă la alta în timpul execuţiei.

CE NU ESTE POLIMORFISMUL

Să nu confundăm polimorfismul cu suprapunerea funcţiilor. De exemplu, următorul program, OVERONLY.CPP, suprapune funcţia *dial* în clasele *rotary* şi *touch_tone*. Deoarece definiţia clasei de bază nu foloseşte cuvântul cheie *virtual*, funcţiile *dial* din clasele derivate vor supradefini funcţia *dial* din clasa de bază. Funcţiile nu sunt, însă, virtuale:

```
#include <iostream.h>
#include <string.h>

class telephone {
  public:
    telephone(char *number, int volume);
    int dial(char *outgoing_number);  ———————— Funcţia nu este virtuală
  protected:
    char phone_number[32];
```



```
    int volume;
};

telephone::telephone(char *number, int volume)
 {
   strcpy(telephone::phone_number, number);
   telephone::volume = volume;
 }

int telephone::dial(char *outgoing_number)
 {
   cout << "Use a rotary or touch-tone phone to call: " <<
     outgoing_number;

   cout << " Volume: " << volume << endl;
   return(1);
 }

class touch_tone: public telephone {
  public:
    touch_tone(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
 {
   cout << "Beep beep beep with touch tone: " << outgoing_number;
   cout << " Volume: " << volume << endl;

   return(1);
 }

class rotary: public telephone {
  public:
    rotary(char *number, int volume) : telephone(number,
      volume) { };
    int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
 {
   cout << "Click click click with rotary: " << outgoing_number;
   cout << " Volume: " << volume << endl;

   return(1);
 }

void main(void)
```

```
{
   touch_tone office("363-1111", 5);
   rotary home("555-1234", 2);

   telephone *cellular = new telephone("111-2222", 3);

   home.dial("222-3333");
   office.dial("333-4444");

   cellular->dial("123-4567");

   cellular = &office;

   cellular->dial("890-1234");
}
```

La compilarea şi execuţia programului, ecranul va afişa următoarele:

```
C:\> OVERONLY  <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Use a rotary or touch-tone phone to call: 123-4567 Volume: 3
Use a rotary or touch-tone phone to call: 890-1234 Volume: 5
```

Cu toate că obiectului *cellular* i s-a atribuit în program adresa obiectului de tip *office*, funcţia membru *dial* nu s-a schimbat ca urmare a acestei atribuiri. Cu alte cuvinte, obiectul nu este polimorfic.

FOLOSIREA FUNCŢIILOR MEMBRU ALE CLASEI DE BAZĂ

În exemplele anterioare, fiecare clasă derivată (*rotary* şi *touch_tone*) au definit funcţia *dial* care a înlocuit funcţia membru virtuală a clasei de bază.

În anumite cazuri, o clasă derivată nu va înlocui funcţia virtuală, ci va folosi chiar funcţia clasei de bază. Să presupunem de exemplu că scrieţi un program de simulare pentru compania Rolls-Royce, care produce motoare de automobile, de camioane şi chiar de avioane. Programul poate folosi clasa *engine*, ca mai jos:

```
class engine {
  public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
  private:
    char name[64];
    int fuel_type;
};
```

Clasa *engine* defineşte funcţia membru *get_fuel* ca o funcţie virtuală (fiecare motor foloseşte un combustibil diferit). În mod prestabilit, funcţia *get_fuel* a clasei *engine* afişează un mesaj care înştiinţează utilizatorul să încarce combustibil obişnuit sau fără plumb, cum se arată în continuare:


```
void engine::get_fuel(int gallons)
{
  cout << "Please add " << gallons << " of regular or unleaded" <<
         endl;
}
```

Pentru un motor de avion, însă, programul va cere utilizatorului să folosească un combustibil special de avion. De asemenea, pentru un camion, programul ar trebui să ceară un combustibil de tip Diesel. În sfârşit, pentru un motor de motocicletă, programul va folosi pur şi simplu funcţia prestabilită sau funcţia clasei de bază. Următorul program, ENGINE.CPP, implementează clasa virtuală *engine*:

```
#include <iostream.h>
#include <string.h>

class engine {
  public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
  private:
    char name[64];
    int fuel_type;
};

engine::engine(char *name, int fuel_type)
{
  strcpy(engine::name, name);
  engine::fuel_type = fuel_type;
}

void engine::get_fuel(int gallons)
{
  cout << "Please add " << gallons << " of regular or unleaded" <<
         endl;
}

class jet_engine : public engine {
  public:
    jet_engine(char *name, int fuel_type) : engine(name,
      fuel_type) {};
    void get_fuel(int gallons)
      {
        cout << "Please add " << gallons << " of JP4 jet fuel" <<
              endl;
      };
};

class truck_engine : public engine {
```

```
public:
  truck_engine(char *name, int fuel_type) : engine(name,
    fuel_type) {};
  void get_fuel(int gallons)
    {
       cout << "Please add " << gallons << " of diesel fuel" <<
               endl;
    };
};


void main(void)
 {
   engine rolls("Rolls Royce", 1);
   jet_engine F100("Jet aircraft", 2);
   truck_engine truck("Delivery truck", 3);
   engine harley("Harley Davidson", 1);

   rolls.get_fuel(20);
   F100.get_fuel(1000);
   truck.get_fuel(40);
   harley.get_fuel(5);
 }
```

Se observă că obiectele *rolls* şi *harley* folosesc funcţia membru a clasei de bază. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> ENGINE  <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of JP4 jet fuel
Please add 40 of diesel fuel
Please add 5 of regular or unleaded
```

Aşa cum probabil aţi observat, programul acesta nu ilustrează polimorfismul - nici un obiect nu îşi schimbă forma.

Următorul program, USER_ENG.CPP, foloseşte obiectul pointer *user_engine* pentru a afişa tipul de combustibil necesar unui anumit tip de motor, selectat de utilizator. Programul începe prin a cere utilizatorului să aleagă tipul de motor. În funcţie de motorul selectat, pointerul *user_engine* va lua una din formele posibile ale unui obiect de tip *engine*:

```
#include <ctype.h>

void main(void)
 {
   engine rolls("Rolls Royce", 1);
   jet_engine F100("Jet aircraft", 2);
   truck_engine truck("Delivery truck", 3);
   engine harley("Harley Davidson", 1);
```



```
   engine *user_engine;

   int not_done = 1;

   while (not_done)
    {
      char user_choice;

      cout << "Select an engine type" << endl;
      cout << "S Standard\t Jet engine\tD Diesel\tQ Quit: " << endl;

      cin >> user_choice;

      user_choice = toupper(user_choice);

      switch (user_choice) {
        case 'S': user_engine = &rolls;
                  break;
        case 'J': user_engine = &F100;
                  break;
        case 'D': user_engine = &truck;
                  break;
        case 'Q': not_done = 0;
                  user_engine = (engine *) 0;
                  break;
        default: user_engine = (engine *) 0;
                  break;
      };

      if (user_engine)
       {
         user_engine->get_fuel(5);
         cout << endl << endl << endl;
       }
    }
 }
```

Observaţie: *Acest program nu conţine definiţiile claselor şi ale funcţiilor membru.*

După cum se observă, programul atribuie un anumit tip de motor pointerului *user_engine*, în funcţie de răspunsul utilizatorului.

Deoarece pointerul *user_engine* poate referi unul din mai multe tipuri de obiecte, el este un obiect polimorfic. Compilatorul nu poate determina în nici un fel la care obiect face pointerul referinţă la un anumit moment. Ca atare, polimorfismul apare la execuţie, folosind suportul dat de compilatorul C++ ce va fi discutat în Capitolul 13.

REGULI PENTRU FUNCȚII POLIMORFICE

Dacă nu precedați funcția membru a clasei de bază prin cuvântul-cheie *virtual*, polimorfismul nu va avea loc. În plus, dacă tipul rezultatului sau tipul parametrilor funcției din clasa derivată (cu același nume ca al funcției virtuale) nu sunt aceiași, polimorfismul nu va avea loc. De exemplu, următorul program, MISMATCH.CPP, modifică tipul parametrului *gallons* din *int* în *long*. Ca urmare, pointerul *engine* nu va fi polimorfic.

```
#include <iostream.h>
#include <string.h>

class engine {
  public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
  private:
    char name[64];
    int fuel_type;                  Definiția int a parametrului gallons
};

engine::engine(char *name, int fuel_type)
 {
   strcpy(engine::name, name);
   engine::fuel_type = fuel_type;
 }

void engine::get_fuel(int gallons)
 {
   cout << "Please add " << gallons << " of regular or unleaded" <<
          endl;
 }

class jet_engine : public engine {
  public:
    jet_engine(char *name, int fuel_type) : engine(name,
      fuel_type) {};
    void get_fuel(long gallons)
                                    Definiția log a parametrului gallons
      {
        cout << "Please add " << gallons << " of JP4 jet fuel" <<
               endl;
      };
};
void main(void)
 {
   engine rolls("Rolls Royce", 1);
   jet_engine F100("Jet aircraft", 2);

   engine *engine_ptr;

   engine_ptr = &rolls;
   engine_ptr->get_fuel(20);

   engine_ptr = &F100;
   engine_ptr->get_fuel(1000);
 }
```

Deoarece parametrii funcțiilor membru *get_fuel* nu concordă în tip, polimorfismul nu apare. La compilarea și execuția programului, pe ecran va apărea:

```
C:\> MISMATCH  <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of regular or unleaded
```

Exersați cu acest program, modificând tipul *long* iarăși în *int*. Acum va apărea polimorfismul, iar programul va afișa:

```
C:\> MISMATCH  <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of JP4 jet fuel
```

FUNCȚII VIRTUALE ȘI MOȘTENIREA PE MAI MULTE NIVELE

În Capitolul 2 ați examinat moștenirea claselor în detaliu. În funcție de structura claselor, apar situații când se folosește moștenirea pe mai multe nivele. De exemplu, să presupunem că vreți să creați o clasă *pay_phone* (telefon public) bazată pe clasa *rotary* prezentată anterior:

```
class pay_phone: public rotary {
  public:
    pay_phone(char *number, int volume, int cost): rotary(number,
      volume) {
      pay_phone::cost = cost;
    }
    int dial(char *outgoing_number);
  private:
    int cost;
};
```

Să presupunem apoi că, clasa *pay_phone* are nevoie de propria funcție *dial*, care cere utilizatorului să introducă o fisă de 100 lei. Se poate declara funcția membru *dial* ca virtuală în clasa *rotary* astfel:

```
class rotary: public telephone {
  public:
```

```
    rotary(char *number, int volume) : telephone(number,
      volume) { };
    virtual int dial(char *outgoing_number);
};
```

Declarând funcţia ca virtuală, programul va putea crea obiecte cu propriile funcţii membru *dial*, dar şi obiecte ce folosesc funcţia *dial* din clasa *rotary*. Următorul program, PAYPHONE.CPP, creează un obiect polimorfic ce poate fi fie *rotary* (telefon cu disc), fie de tipul *pay_phone* (telefon public).

```
#include <iostream.h>
#include <string.h>

class telephone {
  public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
  protected:
    char phone_number[32];
    int volume;
};

telephone::telephone(char *number, int volume)
 {
   strcpy(telephone::phone_number, number);
   telephone::volume = volume;
 }

int telephone::dial(char *outgoing_number)
 {
   cout << "Use a rotary or touch-tone phone to call: " <<
           outgoing_number;

   cout << " Volume: " << volume << endl;
   return(1);
 }

class rotary: public telephone {
  public:
    rotary(char *number, int volume) : telephone(number,
      volume) { };
    virtual int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
 {
   cout << "Click click click with rotary: " << outgoing_number;
   cout << " Volume: " << volume << endl;
   return(1);
 }
```



```
class pay_phone: public rotary {
  public:
    pay_phone(char *number, int volume, int cost): rotary(number,
      volume) {
      pay_phone::cost = cost;
    }
    int dial(char *outgoing_number);
  private:
    int cost;
};

int pay_phone::dial(char *outgoing_number)
 {
   cout << "Please deposit " << cost << " cents" << endl;

   return(rotary::dial(outgoing_number));
 }

void main(void)
 {
    pay_phone street("363-1111", 5, 25);
    rotary home("555-1234", 2);

    telephone *phone;

    phone = &home;
    phone->dial("222-3333");

    phone = &street;
    phone->dial("333-4444");
 }
```

După cum se observă, pointerul *phone* face referinţă la două obiecte de tipuri diferite. La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> PAYPHONE  <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Please deposit 25 cents
Click click click with rotary: 333-4444 Volume: 5
```

Se observă că clasa *pay_phone* înlocuieşte funcţia membru *dial*. Examinaţi cu atenţie cum clasa *pay-phone* defineşte funcţia *dial*:

```
int pay_phone::dial(char *outgoing_number)
 {
   cout << "Please deposit " << cost << " cents" << endl;

   return(rotary::dial(outgoing_number));
 }
```

La început, funcţia înştiinţează utilizatorul să introducă fisa corespunzătoare. Apoi, funcţia foloseşte operatorul de rezoluţie globală (::) pentru a apela funcţia *dial* definită în clasa *rotary*, returnând valoarea returnată de *rotary::dial*. Experimentaţi acest program. Dacă, de exemplu, îndepărtaţi cuvântul cheie *virtual* din faţa funcţiei *rotary::dial*, polimorfismul nu va mai avea loc.

SĂ ÎNŢELEGEM FUNCŢIILE PUR VIRTUALE

Abstractizarea este procesul de ignorare temporară a detaliilor nesemnificative, în vederea concentrării asupra conceptelor importante ale unei probleme. În activitatea de programare, când se creează primele prototipuri ale unor programe mari, se inserează adesea definiţii simple de funcţii, ce vor fi ulterior înlocuite cu definiţiile reale, mai complicate. De exemplu, un astfel de prototip ar putea fi:

```
void health_care_program(void)
  {
     cout << "Health care program function-to be completed later" <<
             endl;
  }
```

Scriind astfel de funcţii simple într-un program, se pot mai uşor depana şi pune la punct anumite părţi ale programului.

La proiectarea claselor unui program, se creează mai întâi clasele de bază, din care vor fi definite clasele ce vor fi utilizate în program. O *clasă abstractă* este o definiţie de clasă al cărei scop este de a pune bazele pentru derivarea altor clase. În mod obişnuit, nu se creează obiecte dintr-o clasă abstractă. În schimb, clasele derivate vor moşteni caracteristicile clasei abstracte.

La folosirea claselor abstracte, apar situaţii când acestea conţin funcţii virtuale ce pot fi folosite sau înlocuite de clasa derivată. În funcţie de definiţia clasei, există situaţii când se defineşte o funcţie virtuală a cărei semnificaţie trebuie indicată de către clasa derivată. Funcţiile care trebuie definite de către clasa derivată se numesc *funcţii pur virtuale*. Pentru a crea o funcţie pur virtuală, aceasta trebuie iniţializată cu zero în clasa abstractă. De exemplu, clasa *abstract_telephone*, prezentată în continuare, defineşte funcţia membru *dial* ca o funcţie pur virtuală:

```
class abstract_telephone {
  public:
    abstract_telephone(char *number, int volume);
    virtual int dial(char *outgoing_number) = 0;  ——— Funcţie pur virtuală
  protected:
    char phone_number[32];
    int volume;
};
```


În cadrul clasei derivate, trebuie specificată o funcţie pentru fiecare funcţie pur virtuală. Următorul program, VIRTDIAL.CPP, foloseşte o funcţie pur virtuală pentru *dial*:

```
#include <iostream.h>
#include <string.h>

class abstract_telephone {
  public:
    abstract_telephone(char *number, int volume);
    virtual int dial(char *outgoing_number) = 0;
  protected:
    char phone_number[32];
    int volume;
};

abstract_telephone::abstract_telephone(char *number, int volume)
  {
    strcpy(abstract_telephone::phone_number, number);
    abstract_telephone::volume = volume;
  }

class touch_tone: public abstract_telephone {
  public:
    touch_tone(char *number, int volume) :
      abstract_telephone(number, volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
  {
    cout << "Beep beep beep with touch tone: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
  }

void main(void)
  {
    touch_tone office("363-1111", 5);
    office.dial("333-4444");
  }
```

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> VIRTDIAL  <ENTER>
Beep beep beep with touch tone: 333-4444 Volume: 5
```

Dacă continuaţi să exersaţi cu acest program şi înlăturaţi funcţia *touch_tone::dial*, programul va fi compilat, dar programul de editare a legăturilor (linkeditare) va afişa mesaje de erori, care arată că funcţia nu va fi găsită. Într-adevăr, clasele derivate trebuie să specifice definiţii pentru funcţiile pur virtuale.



SĂ ÎNȚELEGEM FUNCȚIILE PUR VIRTUALE

Dacă examinați definițiile claselor C++, veți putea întâlni funcții membru virtuale care sunt inițializate cu zero. Astfel de funcții, denumite funcții *pur virtuale*, sunt folosite în clasele abstracte pentru a specifica o funcție pe care clasele derivate din clasa abstractă trebuie să o definească ulterior. Dacă o clasă derivată nu definește o funcție pur virtuală, editorul de legături va afișa mesaje de eroare și nu va construi programul executabil.

REZUMAT

Acest capitol a examinat polimorfismul și funcțiile virtuale în detaliu. Înainte de a trece la Capitolul 9, asigurați-vă că ați învățat următoarele:

- ✓ Polimorfismul este proprietatea unui obiect de a lua două sau mai multe forme diferite în timpul execuției programului. C++ suportă polimorfismul prin folosirea funcțiilor virtuale.
- ✓ O funcție virtuală este un fel de funcție "variabilă", o funcție membru executabilă, ce urmează a fi ulterior înlocuită cu funcții membru derivate. Este precedată de cuvântul-cheie virtual.
- ✓ Când într-un program se derivează o clasă dintr-o clasă de bază ce folosește o funcție virtuală, clasa derivată poate înlocui funcția virtuală cu o funcție proprie sau poate folosi chiar funcția definită în clasa de bază.
- ✓ Când o clasă derivată definește o funcție membru al cărui nume este același cu al funcției din clasa de bază, tipul rezultatului și tipul parametrilor trebuie să fie în concordanță cu tipul rezultatului și al parametrilor funcției din clasa de bază; în caz contrar polimorfismul nu va fi creat.
- ✓ O funcție pur virtuală este o funcție virtuală inițializată de clasa de bază cu 0. Clasa derivată trebuie să furnizeze o definiție pentru o funcție pur virtuală.

CAPITOLUL 9
ACOMODAREA CU REGULILE DOMENIULUI DE VIZIBILITATE

În cadrul programelor, numele identifică variabile, funcții, clase, tipuri ș.a.m.d. *Domeniul* unui nume definește părțile programului în care numele are sens. De exemplu, variabilele cu *domeniu local* sunt vizibile în interiorul funcției în care sunt definite. În mod similar, variabilele cu *domeniu global* sunt vizibile în întreg programul, începând cu locul în care sunt declarate. Așa cum veți învăța în acest capitol, folosirea corectă a domeniului necesită mai mult decât cunoașterea domeniului local sau global al variabilelor. Când veți termina acest capitol, veți înțelege diferite aspecte ale conceptului de vizibilitate în C++. Exersați fiecare program în toate amănuntele. Multe din conceptele subtile ilustrate în programe pot fi sesizate numai după o examinare atentă. La încheierea capitolului, veți cunoaște:

- Ce este domeniul unui nume (identificator)
- Ce sunt domeniul local și domeniul global
- Cum se indică scopul unui identificator
- Care sunt cele 4 tipuri de domeniu
- Ce este domeniul unei funcții
- Ce este domeniul unui fișier
- Ce este domeniul unei clase
- Unde se pot declara variabilele într-un program C++
- Ce se întâmplă când numele unei variabile locale intră în conflict cu numele unei variabile globale sau al unei variabile de clasă
- Ce este durata de viață a unei variabile
- Ce este legarea internă și externă

DECLARAȚII ȘI DEFINIȚII

Când citim cărți sau articole despre C sau C++, termenii *definiție* și *declarație* se folosesc deseori alternativ. Cu toate că par a fi sinonime, sensurile celor două cuvinte diferă. O declarație specifică proprietățile unei variabile: un tip (ca *int*,

float sau *char*) şi un nume. De exemplu, următoarele instrucţiuni sunt declaraţii valide în C sau C++:

```
int age, count;
float salary;
long int miles_to_the_moon;

struct date {
  int day;
  int month;
  int year } birthday;

enum suite { clubs, diamonds, hearts, spades };
```

În general, când o declaraţie are şi efectul alocării de memorie pentru stocarea obiectului, atunci această declaraţie este şi o *definiţie*. Fiecare din declaraţiile anterioare au alocat şi memorie pentru entităţile declarate, deci sunt şi definiţii. Următoarele declaraţii, însă, nu alocă memorie, deci nu sunt definiţii:

```
extern int error_flag;
extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;

int some_function(int, float, char);   // Function prototype
```

În acest caz, declaraţiile specifică tipul şi numele fiecărei variabile. Totuşi, cuvântul cheie *extern* informează compilatorul că variabilele sunt *definite* în altă parte. Prin urmare, compilatorul nu va aloca memorie pentru stocarea variabilelor. Aşa cum arată prototipul funcţiei anterioare, declaraţiile nu sunt caracteristice numai variabilelor. Prototipurile de funcţii sunt de asemenea declaraţii, deoarece introduc în program numele funcţiei. Instrucţiunile funcţiei, pe de altă parte, definesc funcţia.

În sensul cel mai simplu, o declaraţie introduce un nume. Pentru compilatorul C++, fiecare nume are asociat un tip, care determină setul de valori pe care îl poate lua entitatea respectivă şi setul de operaţii ce poate fi realizat pe acea entitate. Domeniul unui nume specifică zona programului unde numele are sens. Datorită domeniului de vizibilitate, acelaşi nume poate fi folosit în mod diferit, în diferite părţi ale programului.

PRINCIPII DE BAZĂ ALE DOMENIULUI DE VIZIBILITATE

Fiecare entitate (variabilă, funcţie, clasă etc.) dintr-un program are un nume. Domeniul numelui entităţii defineşte acea zonă a programului unde numele are sens. C++ defineşte 4 tipuri de domeniu: domeniul local, domeniul funcţiei, domeniul fişierului, domeniul clasei.





SĂ ÎNŢELEGEM DECLARAŢIILE ŞI DEFINIŢIILE

În cadrul unui program, variabilele, clasele, funcţiile şi toate celelalte entităţi au nume. Înainte de a referi (utiliza) o entitate, numele ei trebuie să fie mai întâi declarat. O declaraţie specifică un nume şi un tip, similar celor arătate mai jos:

```
int i, j, k;
float salary;
char string[256];
extern istream_withassign _Cdecl cin;
```

În sensul cel mai simplu, dacă o declaraţie alocă memorie pentru o anumită entitate, devine automat şi o definiţie. În cazul declaraţiilor anterioare, numai ultima instrucţiune (care foloseşte cuvântul cheie *extern*) nu alocă memorie. Prin urmare, acea instrucţiune nu este o definiţie. Mai exact, totuşi, o declaraţie este o definiţie numai dacă nu intră într-una din următoarele categorii:

- declară prototipul unei funcţii
- conţine cuvântul cheie *extern*, dar nu face iniţializări şi nu conţine instrucţiuni din structura unei funcţii
- declară numele unei clase
- declară un membru de clasă statică
- declară un nou tip prin *typedef*

SĂ ÎNŢELEGEM DOMENIUL LOCAL

Domeniul local corespunde acelor entităţi definite într-un bloc al programului (între acolade), ori parametrilor formali dintr-o definiţie de funcţii. Variabilele cu domeniu local sunt cunoscute numai în interiorul blocului în care sunt declarate, începând cu locul declaraţiei. De exemplu, următorul program, LOCAL.CPP, creează 3 variabile locale, denumite *chapter, title* şi *book*:

```
#include <iostream.h>

void main(void)
 {                                                        Declaraţii variabile locale
   int chapter = 9;
   char *title = "Getting Up to Speed with Rules of Scope";
   char *book = "Success with C++";
```

```
    cout << "Chapter " << chapter << " " << title << endl;
    cout << book << endl;
  }
```

În C, variabilele locale pot fi definite la începutul oricărui bloc al programului. În C++, însă, variabilele pot fi definite în orice poziţie, chiar combinate cu instrucţiunile programului. Prin urmare, următorul program, MIXLOCAL.CPP, este identic din punct de vedere funcţional cu programul anterior:

```
#include <iostream.h>

void main(void)
 {
   int chapter = 9;
   char *title = "Getting Up to Speed with Rules of Scope";

   cout << "Chapter " << chapter << " " << title << endl;

   char *book = "Success with C++";
   cout << book << endl;
 }
```

Prin declararea variabilelor în apropierea locului de folosire, se poate îmbunătăţi claritatea programelor mari. Un motiv pentru a declara variabilele în interiorul instrucţiunilor programului este de a putea declara variabila de control în interiorul unui ciclu *for*, ca mai jos:

Declaraţie variabilă locală

```
for (int i = 0; i < 50; i++)
   some_array[i] = 0;
```

Se observă că ciclul *for* declară şi iniţializează variabila *i* în interiorul ciclului. După încheierea ciclului, variabila locală *i* rămâne definită. Următorul program, SHOW_I.CPP, defineşte o variabilă *i* în interiorul ciclului *for*. După terminarea ciclului, programul afişează valoarea variabilei:

```
#include <iostream.h>

void main(void)
 {
   for (int i = 0; i < 25; i++)
     cout << i;

   cout << endl << "Value of i is " << i << endl;
 }
```




DECLARAREA VARIABILELOR ÎN INTERIORUL UNUI BLOC DE INSTRUCŢIUNI

C++ permite programelor să declare variabilele şi între instrucţiuni, nu numai la începutul blocurilor de instrucţiuni. În multe situaţii, se poate îmbunătăţi claritatea unui program dacă variabilele se declară în apropierea locului lor de folosire. De exemplu, următoarele instrucţiuni declară variabilele *age* şi *name* între instrucţiunile programului:

```
cout << "Type in the employee name and age: ";

int age;
char name[64];

cin >> name >> age;
```

În mod similar, următoarea instrucţiune declară variabila *i* într-o instrucţiune *for*:

```
for (int i = 0; i < 10; i++)
   cout << i << endl;
```

Modul de declarare a variabilelor într-un program este, ca şi alte chestiuni de ingineria programelor, discutabil. Ca o regulă, totuşi, dacă declararea unei variabile între instrucţiunile programului îmbunătăţeşte claritatea acestuia, atunci este recomandabilă.

Aşa cum s-a arătat, numele variabilelor locale au sens numai în interiorul blocului în care sunt definite. Dacă programul încearcă să folosească o variabilă într-un bloc înainte de declararea ei, sau în alt bloc disjunct, numele variabilei este necunoscut şi va apărea o eroare de sintaxă. Următorul program, TWOBOOKS.CPP, de exemplu, creează o variabilă locală denumită *book* şi atribuie acesteia un şir de caractere. În interiorul unei instrucţiuni *if*, programul creează o a doua variabilă locală denumită *book*, căreia i se atribuie altă valoare. Deoarece fiecare variabilă are propriul ei domeniu de vizibilitate, cele două variabile vor fi distincte, chiar dacă au acelaşi nume:

```
#include <iostream.h>

void main(void)
 {
   char *book = "Success with C++";   // Prima declaraţie a variabilei book

   if (book)
```

```
    {
      char *book = "Rescued by C++";  ———— A doua declarație a variabilei book

      cout << "Start with the book " << book << endl;
    }

    cout << "Then read the book " << book << endl;
}
```

Observație: *Acest program este prezentat pentru a ilustra domeniul local,* ***nefiind*** *un exemplu bun de programare. De regulă, e bine ca funcțiile să nu aibă două sau mai multe variabile cu același nume. Folosirea aceluiași nume pentru variabile diferite face ca programul să fie greu de înțeles.*

Câteva definiții prezentate în acest capitol folosesc expresia „entitatea poate fi folosită oriunde în blocul în care a fost declarată, începând cu locul declarării". Aceasta înseamnă că, dacă o variabilă este declarată în mijlocul unui bloc, ea nu va putea fi referită anterior declarației. Variabila va fi cunoscută (în interiorul blocului respectiv) numai după declarare. Următoarele instrucțiuni vor genera, prin urmare, o eroare de sintaxă:

```
cout << "This is chapter " << chapter;
int chapter = 9;
```

Eroarea apare deoarece programul încearcă să folosească variabila *chapter* înainte de a fi fost declarată.

SĂ ÎNȚELEGEM DOMENIUL LOCAL

Variabilele cu domeniu local sunt cunoscute numai în blocul în care sunt declarate, începând cu locul declarației. C++ permite programelor să declare variabile locale în orice parte a programului, chiar intercalate cu instrucțiunile programului. Declarând variabilele aproape de locul folosirii lor, se poate îmbunătăți claritatea programelor. C++ consideră parametrii formali ai funcțiilor ca variabile locale, care sunt cunoscute numai în interiorul funcției.

SĂ ÎNȚELEGEM DOMENIUL UNEI FUNCȚII

O etichetă definește o locație situată între instrucțiunile unei funcții. Folosind instrucțiunea *goto*, programul poate transfera controlul instrucțiunii ce urmează după etichetă. Următorul program, GOTOLOOP.CPP, folosește instrucțiunea *goto* și eticheta *loop* pentru a afișa pe ecran numerele naturale între 1 și 50:

```
#include <iostream.h>

void main(void)
```



```
{
  int i = 1;

  loop:
    cout << i++ << endl;

    if (i <= 50)
      goto loop;
}
```

La folosirea instrucțiunii *goto*, eticheta la care se face transferul trebuie să se afle în interiorul aceleiași funcții ca și *goto*. Cu alte cuvinte, etichetele au domeniu de funcție. Numele etichetelor sunt singurele entități care au un astfel de domeniu.

SĂ ÎNȚELEGEM DOMENIUL UNEI FUNCȚII

Domeniul unei funcții restricționează valabilitatea unui identificator în interiorul funcției în care este declarat. Numele de etichete sunt singurele entități în C++ care au domeniu de funcție. Prin urmare, instrucțiunea *goto* nu poate efectua transferul de la o funcție la o etichetă definită în altă funcție.

SĂ ÎNȚELEGEM DOMENIUL UNUI FIȘIER

Variabilele declarate într-un fișier sursă, dar în afara programului, sunt variabile cu *domeniu fișier*. În sensul cel mai simplu, variabilele cu domeniu fișier sunt variabile globale. Variabilele globale sunt recunoscute în toate blocurile programului care urmează după declarație. Următorul program, GLOBAL.CPP, declară variabilele *book* și *chapter* ca variabile globale. Programul afișează apoi valoarea fiecărei variabile în două funcții diferite.

```
#include <iostream.h>

char *book = "Success with C++";
char *chapter = "Getting Up to Speed with Scope";

void show_globals(int count)
 {
   for (int i = 0; i < count; i++)
     cout << book << " " << chapter << endl;
 }

void main(void)
 {
   cout << "This book: " << book << endl;
   cout << "Chapter 13: " << chapter << endl;
```

```
    show_globals(1);
}
```

Reamintim că programele nu pot utiliza o variabilă globală înainte de a fi declarată. Următoarele instrucţiuni, de exemplu, încearcă să folosească variabila globală *error_member* în funcţia *show_error*, înainte ca variabila să fie declarată. La compilarea acestor instrucţiuni va apărea o eroare de sintaxă:

```
void show_error(void)
{
    cerr << "Error in processing # " << error_number << endl;
}

int error_number;  // Global variable
```

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM DOMENIUL UNUI FIŞIER

Domeniul unui fişier specifică faptul că variabilele declarate în afara tuturor blocurilor programului sunt recunoscute în toate blocurile care urmează după declaraţia variabilelor. Variabilele având domeniul fişier sunt globale. Deoarece variabilele globale sunt uşor accesibile în întregul program, ele sunt foarte susceptibile la erori. De aceea, de regulă, programele trebuie să restrângă la maximum folosirea variabilelor globale.

SĂ ÎNŢELEGEM DOMENIUL UNEI CLASE

Clasele asigură camuflajul datelor şi încapsularea. În interiorul unei clase, membrii acesteia sunt locali şi pot fi folosiţi numai de funcţiile membru ale clasei (sau de funcţiile din clasele derivate), cu operatorul punct (.) sau de indirectare (→), după operatorul de rezoluţie (::), sau după clasele declarate ca *friend*. Această secţiune examinează diferite operaţii cu membrii clasei, care dispun de domeniu clasă.

SĂ ÎNŢELEGEM MEMBRII DE TIP PUBLIC ŞI PRIVATE AI UNEI CLASE

Aşa cum aţi citit în Capitolul 2, membrii unei clase pot fi de tip *public* sau *private*, în funcţie de modul în care dorim ca programul să aibă acces la variabile. Membrii de tip *private* pot fi folosiţi numai de funcţiile membru ale clasei (sau funcţiile claselor derivate sau funcţiile *friend*). Membrii de tip *public*, pe de altă parte, pot fi apelaţi din orice parte a programului folosind operatorul punct (.) sau de indirectare (→). De exemplu, următorul program, PUBPRIV.CPP, crează o clasă *Book*, care conţine variabilele membru *title, author* şi *pages*. Clasa conţine, de asemenea, funcţia constructor *Book* şi funcţia membru *show_book*:

```
class Book {
 public:
```



```
   Book(char *book_title, char *book_author, int book_pages)
     {
       strcpy(title, book_title);
       strcpy(author, book_author);
       pages = book_pages;
     };

   void show_book(void)
     {
       cout << "Book: " << title << endl;
       cout << "Author: " << author << endl;
       cout << "Pages: " << pages << endl;
     }
 private:
   char title[64];
   char author[64];
   int pages;
};
```

După cum se vede, funcţiile membru ale clasei sunt implementate folosind instrucţiuni inline. Definind variabilele membru de tip *private*, programul restricţionează accesul la acestea numai la funcţiile membru ale clasei. Următoarele instrucţiuni implementează programul PUBPRIV.CPP:

```
#include <iostream.h>
#include <string.h>

class Book {
 public:
   Book(char *book_title, char *book_author, int book_pages)
     {
       strcpy(title, book_title);
       strcpy(author, book_author);
       pages = book_pages;
     };

   void show_book(void)
     {
       cout << "Book: " << title << endl;
       cout << "Author: " << author << endl;
       cout << "Pages: " << pages << endl;
     }
 private:
   char title[64];
   char author[64];
   int pages;
};

void main(void)
 {
   Book cpp_book("Success with C++", "Jamsa", 528);
   cpp_book.show_book();
 }
```

Aşa cum se observă, clasa *Book* nu permite accesul programului la variabilele membru, definindu-le de tip *private*. Singurul mod de a avea acces la variabilele membru este prin intermediul funcţiilor membru ale clasei de tip *public*, cum sunt funcţia constructor sau funcţia *show_book*.

CHEIA SUCCESULUI

FOLOSIREA MEMBRILOR DE TIP PUBLIC ŞI PRIVATE AI UNEI CLASE

Când creaţi propriile clase, cuvintele cheie *private* şi *public* permit programului să controleze accesul la membri, asemănător domeniului variabilelor. De regulă, se declară variabilele membru de tip *private* cât mai des posibil, permiţând accesul la variabile numai prin funcţiile membru. De asemenea, numai acele funcţii pe care programul le foloseşte direct trebuie să fie de tip *public*. Cu alte cuvinte, folosiţi teoria „trebuie să ştiu" pentru a determina care membri trebuie să fie de tip *public* şi care de tip *private*.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM ACCESUL CONTROLAT

Când citiţi despre avantajele de a defini variabilele clasei de tip *private* întâlniţi şi sintagma *acces controlat*. Când variabilele membru sunt de tip *public*, programul poate schimba variabilele oricând şi oricum. Să presupunem, totuşi, că aţi creat o clasă ce modelează un reactor nuclear. Una dintre variabilele membru e denumită *protecţie_la_scurgere*. Când variabila conţine valoarea 1, reactorul nuclear este protejat împotriva scurgerilor. Când variabila conţine valoarea 0, se presupune că reactorul e decuplat şi că protecţia poate fi dezactivată. Dacă variabila membru este de tip *public*, un programator ar putea intenţionat să schimbe valoarea variabilei folosind o instrucţiune de tipul:

```
reactor.protectie_la_scurgere = 0;
```

De asemenea, programatorul ar putea accidental să schimbe valoarea variabilei printr-o eroare de programare. De exemplu, următoarea instrucţiune *if* se presupune că verifică dacă valoarea variabilei este 0. Din păcate, deoarece programatorul a folosit operatorul de atribuire (=) în locul testului la egalitate (==), variabilei membru i se atribuie valoarea 0:

```
if (reactor.protectie_la_scurgere = 0)
```

Pentru a evita astfel de erori, variabila ar trebui folosită numai prin intermediul funcţiilor membru, ca *setează_protecţia*, care poate cere utilizatorului o parolă, sau *obţine_protecţia*, care returnează valoarea curentă a variabilei. Acesta este *accesul controlat*. Când scrieţi un program, cunoaşteţi cum trebuie folosită fiecare variabilă membru. Declarând acele variabile de tip *private*, vă puteţi asigura că ele vor fi folosite aşa cum aţi dorit.

Cu domeniul clasei, variabilele membru ale clasei pot fi direct apelate de funcţiile membru. De exemplu, să considerăm fie constructorul *Book*, fie funcţia membru *show_book* prezentate anterior:

```
Book(char *book_title, char *book_author, int book_pages)
  {
    strcpy(title, book_title);
    strcpy(author, book_author);
    pages = book_pages;
  };

void show_book(void)
  {
    cout << "Book: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
  }
```

Se observă că ambele funcţii referă direct variabilele membru prin nume. În cazul funcţiei constructor *Book*, aceasta atribuie funcţiilor membru valorile parametrilor. Când funcţiile membru primesc parametri, sunt situaţii când numele unui parametru poate coincide cu acela al unei variabile membru. De exemplu, să presupunem că funcţia constructor *Book* a folosit numele de parametri *title*, *author* şi *price*, ca mai jos:

```
Book(char *title, char *author, int pages)
```

Când numele variabilelor locale (reţineţi că parametrii formali sunt variabile locale) sunt aceleaşi cu ale membrilor clasei, membrii clasei devin *ascunşi*. Operaţiile din corpul funcţiei care referă numele variabilei vor fi aplicate variabilei locale, nu variabilei membru. În astfel de situaţii, funcţia poate referi direct funcţiile membru folosind operatorul de rezoluţie globală (::), ca mai jos:

```
Book(char *title, char *author, int pages)
     {
       strcpy(Book::title, title);
       strcpy(Book::author, author);
       Book::pages = pages;
     };
```

Următorul program, UNHIDE.CPP, foloseşte operatorul de rezoluţie globală pentru a folosi variabile membru ce au fost ascunse de variabilele locale:

```
#include <iostream.h>
#include <string.h>

class Book {
 public:
   Book(char *title, char *author, int pages)
```

```
   {
     strcpy(Book::title, title);
     strcpy(Book::author, author);
     Book::pages = pages;
   };

  void show_book(void)
   {
     cout << "Book: " << title << endl;
     cout << "Author: " << author << endl;
     cout << "Pages: " << pages << endl;
   }
 private:
   char title[64];
   char author[64];
   int pages;
};

void main(void)
 {
   Book cpp_book("Success with C++", "Jamsa", 528);

   cpp_book.show_book();
 }
```

În cazul programului precedent, clasa *Book* a folosit funcţii inline. Aşa cum aţi citit în Capitolul 2, când o clasă foloseşte funcţii inline, fiecare obiect primeşte câte un exemplar din codul funcţiei. Cu alte cuvinte, programul nu beneficiază de partajarea instrucţiunilor de către mai multe obiecte. Următorul program, NOINLINE.CPP, deplasează funcţiile clasei în afara definiţiei acesteia:

```
#include <iostream.h>
#include <string.h>

class Book {
 public:
   Book(char *title, char *author, int pages);
   void show_book(void);
 private:
   char title[64];
   char author[64];
   int pages;
};

Book::Book(char *title, char *author, int pages)
 {
   strcpy(Book::title, title);
   strcpy(Book::author, author);
   Book::pages = pages;
 };
```

Funcţia constructor a clasei **Book**


```
void Book::show_book(void)
 {
   cout << "Book: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Pages: " << pages << endl;
 }

void main(void)
 {
   Book cpp_book("Success with C++", "Jamsa", 528);

   cpp_book.show_book();
 }
```

Funcţie membru a clasei **Book**

După cum se observă, programul foloseşte operatorul de rezoluţie globală cu numele clasei *Book* pentru a avea acces la funcţiile membru ale clasei. Pe măsură ce obiectele dintr-un program devin mai complexe, poate apărea o coincidenţă de nume între funcţiile membru ale unei clase şi funcţiile membru ale altei clase. Aceste conflicte de nume se pot rezolva folosind operatorul de rezoluţie globală precedat de numele clasei.



ATENŢIE LA MEMBRII ASCUNŞI

La transferul parametrilor către funcţiile membru ale clasei, este posibil ca numele parametrilor (variabilelor locale) să coincidă cu cei ai membrilor clasei. Când apar astfel de conflicte, C++ ascunde membrul clasei, asociind toate referinţele către numele de conflict la variabila locală (parametru). Pentru a putea referi, în astfel de situaţii, membrul clasei, trebuie folosit operatorul de rezoluţie globală (::). De exemplu, următoarea funcţie, *set_salary*, atribuie valoarea specificată în parametrul *salary* variabilei membru *salary* din clasa *employee*:

```
void employee::set_salary(float salary)
 {
   employee::salary = salary;
 }
```

Aşa cum se observă, funcţia foloseşte operatorul de rezoluţie globală cu numele clasei pentru a putea avea acces la variabila membru ascunsă.

SĂ ÎNŢELEGEM CLASELE FRIEND

Aşa cum aţi învăţat, membrii de tip *private* ai unei clase permit un acces controlat asupra clasei respective. Când construiţi programe cu obiecte dependente, claritatea programului sau camuflajul informaţiei se pot îmbunătăţi prin acordarea drepturilor de acces la unul sau mai mulţi membri de tip *private* ai clasei

unei anumite clase, denumită *friend* (prietenă). Să presupunem, de exemplu, că un program foloseşte clasa *employee*, care conţine membri de tip *public* şi *private*, cum se arată mai jos:

```
class employee {
  public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(void);
  private:
    char name[256];
    float salary;
    char ssan[256];
};
```

Apoi, să presupunem că în program există şi clasa *manager* care trebuie să aibă acces direct la membrii de tip *private* ai clasei *employee*. Una din soluţii ar fi să declarăm de tip *public* toţi membrii clasei *employee*, pentru a-i face accesibili funcţiilor din clasa *manager*. Dar, din păcate, aceştia ar deveni accesibili în întregul program. O soluţie bună ar fi ca membrii să fie accesibili numai clasei *manager*, specificând clasa *manager* ca fiind *friend* al clasei *employee*:

```
class employee {
  public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(void);
    friend manager; ————————— Specificare drept friend a obiectelor
  private:                    din clasa manager
    char name[256];
    float salary;
    char ssan[256];
};
```

Următorul program, MGR_UPD.CPP, creează obiectele *employee* şi *manager*. Programul transmite apoi obiectul *employee* funcţiei membru *update_employee* a clasei *manager*, care actualizează obiectul. Pentru a permite funcţiei să schimbe obiectul *employee*, programul transmite funcţiei obiectul prin adresă:

```
#include <iostream.h>
#include <string.h>

class manager;

class employee {
  public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(char *name, float salary, char *ssan);
    friend manager;
  private:
```



```
    char name[256];
    float salary;
    char ssan[256];
};

employee::employee(char *name, float salary, char *ssan)
 {
   strcpy(employee::name, name);
   strcpy(employee::ssan, ssan);
   employee::salary = salary;
 }

void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Salary: " << salary << endl;
   cout << "SSAN: " << ssan << endl;
 }

void employee::change_employee(char *name, float salary, char *ssan)
 {
   strcpy(employee::name, name);
   strcpy(employee::ssan, ssan);
   employee::salary = salary;
 }

class manager {
  public:
    void update_employee(employee *emp, char *name, float salary,
      char *ssan);
    // Other members
};

void manager::update_employee(employee *emp, char *name,
  float salary, char *ssan)
 {
   emp->change_employee(name, salary, ssan);
 }

void main(void)
 {
   employee worker("Jamsa", 25000.0, "538-66-5444");

   manager the_boss;

   worker.show_employee();

   the_boss.update_employee(&worker, "Jones", 33456.0,
     "111-22-3333");

   worker.show_employee();
 }
```

Să observăm declarația clasei *manager* la începutul programului:

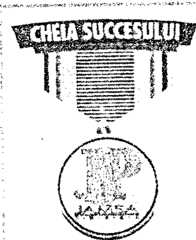
```
class manager;
```

Programul foloseşte această declaraţie pentru a permite obiectului *employee* să refere clasa drept *friend*. Aşa cum se vede, programul transmite obiectul *employee* funcţiei *update_employee* prin adresă, ceea ce permite programului să modifice variabilele membru ale obiectului. Exersaţi cu acest program şi îndepărtaţi instrucţiunea *friend* din declaraţia clasei *employee*. Vor apărea erori de sintaxă, deoarece accesul obiectului *manager* la funcţia *change_employee* nu va mai fi permis.

Declarând clasa *manager* ca *friend*, programul anterior a oferit obiectelor *manager* acces deplin la membrii obiectului *employee*. Din păcate, declarând clasa *manager* ca *friend*, obiectele de tip *manager* pot avea acces complet la membrii obiectului *employee*, în orice mod. În multe cazuri este de dorit să reducem accesul unei clase *friend* la anumite funcţii. De exemplu, următoarea declaraţie de clasă *employee* acordă drepturi de acces numai funcţiei *update_employee* a clasei *manager*:

```
class employee {
  public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(char *name, float salary, char *ssan);
    friend void manager::update_employee(employee *emp, char *name,
      float salary, char *ssan);
  private:
    char name[256];
    float salary;
    char ssan[256];
};
```

Restrângând accesul în acest mod se poate controla mai axeact accesul programului la obiecte, facilitând înţelegerea programului şi reducând şansele de erori.



ACCESUL SPECIAL AL CLASELOR FRIEND LA MEMBRII UNUI OBIECT

Dacă numărul de obiecte într-un program creşte, se poate atunci îmbunătăţi performanţa sau claritatea programului printr-o clasă de obiecte ce au acces special la membrii unui obiect. Dacă într-o declaraţie de clasă declarăm o altă clasă ca fiind prieten (*friend*) al obiectului, atunci oferim clasei *friend* accesul complet la membrii de tip *private* ai obiectului. Următoarea clasă, *comedian*, de exemplu, declară obiectele clasei *agent* ca fiind de tip *friend*. În acest fel, obiectele *agent* au acces la membrii de tip *private* ai clasei *comedian*.


```
class agent;
class comedian {
   public:
        comedian(char *name, char *joke);
        void show_comedian(void);
        friend agent;
  private:
        char *name[64];
        char *best_joke[256];
};
```

De observat declaraţia simplă a clasei *agent* ce precede declaraţia clasei *comedian*. În felul acesta, clasa *comedian* poate referi clasa *agent* înainte ca aceasta din urmă să fie declarată formal. În unele cazuri este necesară reducerea accesului clasei *friend* numai la una dintre funcţiile sale membru. Pentru aceasta, se specifică funcţia în cadrul declaraţiei clasei, ca mai jos:

```
class comedian {
   public:
        comedian(char *name, char *joke);
        void show_comedian(void);
        friend void agent::pay_comic(void);
  private:
        char *name[64];
        char *best_joke[256];
};
```

În acest caz numai funcţia membru *pay_comic* a clasei *agent* are acces la variabilele membru de tip *private* ale clasei *comedian*.

SĂ ÎNŢELEGEM MEMBRII DE TIP PROTECTED

În Capitolul 4 s-a discutat moştenirea în detaliu. Aşa cum aţi învăţat, la derivarea unei clase dintr-o clasă de bază, clasa derivată va avea acces la membrii de tip *public* ai clasei de bază, dar nu şi la membrii de tip *private*. Pentru a oferi claselor derivate un acces de nivel mediu, C++ admite *membri protejaţi* (*protected members*) ai unei clase. Membrii protejaţi pot fi apelaţi de clasele derivate, dar nu şi de restul programului. Următorul program, PROTECT.CPP, derivează clasa *document* folosind clasa de bază *file*:

```
#include <iostream.h>
#include <string.h>

class file {
  public:
    file(char *name, int type);
  protected:
    void show_file_info(void);
    int attributes;
  private:
```

```
    char name[64];
    int type;
};

file::file(char *name, int type)
 {
   strcpy(file::name, name);
   file::type = type;
 }

void file::show_file_info(void)
 {
   cout << "Filename: " << name << endl;
   cout << "Type: " << type << endl;
 }

class document : public file {
  public:
    document(char *name, int type, int attributes);
    void show_document(void);
    void print_document(void) { /* statements */ };
    void save_document(void) { /* statements */ };
  private:
    int error_status;
};

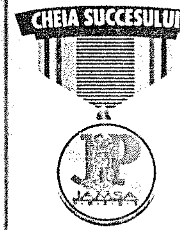
document::document(char *name, int type, int attributes) :
   file(name, type)
 {
   document::attributes = attributes;
 }

void document::show_document(void)
 {
   show_file_info();
   cout << "Attributes: " << attributes << endl;
 }

void main(void)
 {
   document spreadsheet("WORKSHEET.DAT", 1, 255);

   spreadsheet.show_document();
 }
```

După cum se observă, clasa *file* declară membrii *attributes* şi *show_file_info* de tip *protected*. Exersaţi cu acest program, încercând să apelaţi cei doi membri direct din funcţia *main*. Veţi vedea că singurul mod de a avea acces la membrii protejaţi este prin funcţiile membru ale claselor *file* şi *document*.



FOLOSIREA MEMBRILOR DE TIP PROTECTED

La folosirea conceptului de moştenire din C++ pentru derivarea unei clase dintr-o altă clasă de bază, este posibil ca unii membri ai clasei de bază să poată fi utilizaţi de obiectele din clasa derivată, dar nu şi din restul programului. În astfel de situaţii, se pot folosi membrii de tip *protected*. Aceştia sunt accesibili pentru funcţiile membru ale clasei de bază sau ale clasei derivate, dar nu şi din afara acestor funcţii.

EFECTUL CALIFICATORILOR ASUPRA DOMENIULUI

Aşa cum ştiţi, la declararea variabilelor, C++ permite adăugarea unor *calificatori* la declararea tipurilor, ca *long, short* şi alţii. În plus, puteţi preceda declararea variabilelor cu *specificatorii clasei de memorie*, ca *static, auto, extern* sau *register*. Această secţiune examinează specificatorii *extern* şi *static* şi influenţa lor asupra domeniului variabilelor, funcţiilor şi al membrilor clasei. Veţi vedea că, folosind aceşti calificatori, se poate controla accesul la diferiţi identificatori.

FOLOSIREA CALIFICATORULUI EXTERN

Aşa cum aţi învăţat, diferenţa între declaraţii şi definiţii rezidă în alocarea spaţiului de memorie. Dacă precedaţi o declaraţie de cuvântul cheie *extern*, înştiinţaţi compilatorul C++ că folosiţi un identificator care a fost definit în altă parte (în fişierul sursă al programului, în fişierul antet sau într-un fişier de bibliotecă). Următorul program, EXTERN.CPP, foloseşte variabila *title*, care este definită în afara programului:

```
#include <iostream.h>

void show_title(void);

void main(void)
 {
   extern char *title;

   cout << "The title is " << title << endl;

   show_title();
 }
```

După cum se vede, programul declară variabila *title* ca de tip *extern*. În acest fel, compilatorul C++ nu alocă spaţiu pentru variabilă. De fapt, variabila *title* este definită în fişierul sursă SHOWTITL.CPP, care defineşte şi funcţia *show_title*:

```
#include <iostream.h>

char *title = "Success with C++";

void show_title(void)
 {
   cout << "The title of this book is " << title << endl;
 }
```

În cazul că folosiţi compilatorul Borland C++, programul se poate compila astfel:

```
C:\> BCC  EXTERN.CPP  SHOWTITL.CPP  <ENTER>
```

La execuţia programului, pe ecran va apărea:

```
C:\> EXTERN  <ENTER>
The title is Success with C++
The title of this book is Success with C++
```

Exersaţi cu acest program, înlăturând eventual cuvântul *extern* din faţa declaraţiei variabilei *title*, sau mutând declaraţia acesteia în funcţia *show_title*. În ambele cazuri, schimbările efectuate vor duce la erori de sintaxă. Variabilele externe amplifică problemele introduse de variabilele globale, deoarece pot fi schimbate în multe moduri, unele mai puţin evidente pentru programatorul care examinează codul. Poate chiar mai rău este faptul că, deoarece schimbările pot fi operate din diferite fişiere sursă, acestea pot fi detectate foarte greu.

Pentru a reduce vulnerabilitatea unei variabile globale, se poate reduce accesul la aceasta numai la fişierul sursă în care aceasta este definită. În acest caz, declaraţia variabilei este precedată de specificatorul *static*. De exemplu, următoarea instrucţiune restrânge domeniul variabilei *title* la fişierul SHOWTITL.CPP:

```
static char *title = "Succes cu C++";
```

Dacă adăugaţi specificatorul *static* declaraţiei variabilei *title* din fişier şi apoi recompilaţi cele două programe, editorul de legături va afişa un mesaj de eroare despre identificatorul *title* nerezolvat.

INIŢIALIZAREA UNEI VARIABILE EXTERNE

Când o variabilă este precedată de cuvântul *extern*, avem o declaraţie, nu o definiţie, deoarece compilatorul nu alocă memorie pentru variabilă. Aceasta se întâmplă numai în cazul când nu se iniţializează variabila. De exemplu, următorul program, NOEXTERN.CPP, iniţializează variabila externă *title* folosind şirul "Salvat de C++".

Datorită iniţializării, compilatorul va aloca memorie pentru variabilă. La compilarea şi legarea acestui program cu SHOWTITL.CPP, se va afişa:

```
C:\> NOEXTERN  <ENTER>
The title is Rescued by C++
The title of this book is Success with C++
```

După cum se vede, programul foloseşte două variabile *title* diferite. În funcţie de editorul de legături folosit, se pot detecta identificatori dublu definiţi.

SĂ ÎNŢELEGEM MEMBRII DE TIP STATIC AI UNEI CLASE

Aţi învăţat că, la declararea obiectelor unei clase, fiecare obiect îşi preia propriile sale variabile membru. De exemplu, dacă un program foloseşte două sau mai multe obiecte de tipul *date*, fiecare obiect va avea propriile sale variabile *month*, *day* şi *year*. În funcţie de scopul programului, pot apărea situaţii când două sau mai multe obiecte partajează aceeaşi variabilă membru. De exemplu, să presupunem că aveţi obiecte de tipul *Nuke*, fiecare dintre ele destinat detecţiei unei anumite situaţii într-un reactor nuclear. Apoi, să presupunem că atunci când apare o condiţie de eroare, obiectul setează variabila membru *melt_down* la 1. Prin partajarea variabilei între obiectele *Nuke*, toate obiectele programului pot fi simultan la curent cu o situaţie periculoasă. Pentru a partaja o variabilă membru între mai multe obiecte, trebuie precedată declaraţia acelei variabile de cuvântul cheie *static*:

```
class Nuke {
 public:
   Nuke(char *name, int limit);
   void show_status(void);
   void set_melt_down(void);
 private:
   static int melt_down;
   char name[64];
   int limit;
};
```

Obiectele de tipul *Nuke* vor partaja variabilele membru *melt_down*, dar vor avea fiecare copii ale variabilelor membru *name* şi *limit*. În acest exemplu, variabila membru este declarată *private*, dar C++ suportă pe deplin variabile membru de tip *private* şi *public* partajate. Apoi, după declaraţia clasei, variabila membru statică trebuie redeclarată şi eventual iniţializată astfel:

```
int Nuke::melt_down = 0;
```

Următorul program, MELTDOWN.CPP, ilustrează folosirea variabilei membru partajate:

```
#include <iostream.h>
#include <string.h>
```

```
class Nuke {
 public:
   Nuke(char *name, int limit);
   void show_status(void);
   void set_melt_down(void);
 private:
   static int melt_down;
   char name[64];
   int limit;
};

int Nuke::melt_down = 0;

Nuke::Nuke(char *name, int limit)
 {
   strcpy(Nuke::name, name);

   Nuke::limit = limit;
 }

void Nuke::show_status(void)
 {
   cout << "Name: " << name;
   cout << " limit: " << limit;
   cout << " meltdown: " << melt_down << endl;
 }

void Nuke::set_melt_down(void)
 {
    melt_down = 1;
 }

void main(void)
 {
   Nuke rods("Check Rods", 1001);
   Nuke fission("Check Fission", 2002);

   rods.show_status();
   fission.show_status();

   rods.set_melt_down();

   rods.show_status();
   fission.show_status();
 }
```

La compilarea şi execuţia acestui program, pe ecran se va afişa:

```
C:\> MELTDOWN  <ENTER>
Name: Check Rods limit: 1001 meltdown: 0
Name: Check Fission limit: 2002 meltdown: 0
```



```
Name: Check Rods limit: 1001 meltdown: 1
Name: Check Fission limit: 2002 meltdown: 1
```

După cum se observă, când obiectul *rods* determină valoarea variabilei partajate *melt_down* la 1, obiectul *fission* observă imediat schimbarea.



PARTAJAREA UNEI VARIABILE MEMBRU A UNUI OBIECT

Există situaţii când două sau mai multe obiecte trebuie să partajeze aceeaşi variabilă membru. O astfel de variabilă membru partajată se poate crea scriind în faţa acesteia cuvântul cheie *static*. De exemplu, în clasa *Employee* prezentată mai jos, obiectele partajează variabilele *company_name* şi *health_plan*:

```
class Employee {
    public:
        Employee(char *name, long identifier, int job);
        void show_employee(void);
    private:
        char name[64];
        long identifier;
        int job;
        static char company_name[64];
        static char health_plan[25];
};
```

Apoi, în afara declaraţiei clasei, trebuie specificate declaraţiile pentru variabilele partajate, ca mai jos:

```
char Employee::company_name[64];
char Employee::health_plan[25];
```

Fiecare obiect de tipul *Employee* creat de program va partaja aceste două variabile.

FUNCŢII MEMBRU ÎN INTERIORUL ŞI ÎN EXTERIORUL DEFINIŢIEI CLASEI

La declararea unei clase, funcţiile membru pot fi declarate ca *funcţii inline*, când instrucţiunile lor apar în interiorul declaraţiei clasei, sau în afara acestei declaraţii. De exemplu, programul anterior MELTDOWN.CPP a folosit funcţii membru în afara definiţiei clasei, cum este constructorul *Nuke* de mai jos:

```
Nuke::Nuke(char *name, int limit)
 {
```

```
   strcpy(Nuke::name, name);
   Nuke::limit = limit;
 }
```

Funcţia constructor s-ar fi putut declara şi *inline*, astfel:

```
class Nuke {
 public:
   Nuke(char *name, int limit) {
      strcpy(Nuke::name, name);
      Nuke::limit = limit;
    }
  void show_status(void);
   void set_melt_down(void);
 private:
   static int melt_down;
   char name[64];
   int limit;
};
```

Când se declară o funcţie *inline*, fiecare obiect va primi un exemplar al codului funcţiei. La funcţii declarate în afara clasei, pe de altă parte, fiecare obiect partajează aceeaşi copie a funcţiei. În majoritatea cazurilor, se recomandă a se folosi avantajul partajării codului pentru a reduce necesarul de memorie a programului.



PARTAJAREA CODULUI UNEI FUNCŢII MEMBRU

La declararea funcţiilor membru ale clasei, există două posibilităţi. Se pot declara funcţii în cadrul definiţiei clasei (ca funcţii inline) sau în afara acesteia. Avantajul declarării instrucţiunilor unei funcţii membru în afara definiţiei clasei constă în faptul că fiecare obiect nou creat partajează acelaşi exemplar din codul programului. Prin urmare, dacă sunt create 100 de obiecte, se creează numai o singură funcţie, pe care obiectele o vor partaja. În acest fel se reduce şi necesarul de memorie a programului.

SĂ REVEDEM OPERATORUL DE REZOLUŢIE GLOBALĂ

Este posibil ca într-un program numele variabilelor locale să coincidă cu cel al variabilelor globale. În astfel de situaţii, C++ asociază toate referinţele la numele variabilei locale. De exemplu, următorul program, USELOCAL.CPP, defineşte variabila globală *number*, atribuind variabilei valoarea 1001. În funcţia *main*, programul foloseşte variabila locală *number* în cadrul unui ciclu *for* producând un conflict de nume. La afişarea valorii variabilei *number*, programul va afişa valoarea variabilei locale, ascunzând astfel variabila globală.


```
#include <iostream.h>

int number = 1001;  // Declare global variable

void main(void)
 {
   for (int number = 1; number < 5; number++)
     cout << number << endl;

   cout << "Value of number is " << number << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> USELOCAL  <ENTER>
1
2
3
4
Value of number is 5
```

După cum se observă, programul a folosit variabila locală *number*, ignorând variabila globală. Dacă doriţi totuşi folosirea variabilei globale, puteţi scrie înaintea numelui variabilei operatorul de rezoluţie globală (::). Următorul program, USEGLOB.CPP, foloseşte operatorul de rezoluţie globală pentru a afişa valoarea variabilei globale *number* după terminarea ciclului *for*:

```
#include <iostream.h>

int number = 1001;  // Declare global variable

void main(void)
 {
   for (int number = 1; number < 5; number++)
     cout << number << endl;

   cout << "Value of number is " << ::number << endl;
 }
```

La compilarea şi execuţia programului, pe ecran vor apărea rezultatele:

```
C:\> USEGLOB  <ENTER>
1
2
3
4
Value of number is 1001
```

Aşa cum se observă, programul afişează valoarea variabilei globale *number* la sfârşitul ciclului. Când se foloseşte operatorul de rezoluţie globală în acest mod, C++ va referi întotdeauna variabila globală.

Să considerăm, de exemplu, următorul program, WHICHNUM.CPP, care foloseşte 3 variabile diferite denumite *number*:

```
#include <iostream.h>

int number = 1001;  // Global variable

void main(void)
 {
   for (int number = 1; number < 5; number++)
    {
      int number = 1;

      cout << "local number " << number << " ::number is " <<
              ::number << endl;
    }
 }
```

Se observă că în blocul de adâncime maximă programul foloseşte operatorul de rezoluţie globală. Variabila referită cu acest operator este variabila globală, şi nu variabila *number* definită în următorul ciclu exterior blocului. La compilarea şi execuţia acestui program, ecranul va afişa următoarele:

```
C:\> WHICHNUM  <ENTER>
local number 1 ::number is 1001
local number 1 ::number is 1001
local number 1 ::number is 1001
local number 1 ::number is 1001
```

Aşa cum probabil aţi observat, folosirea variabilelor cu acelaşi nume în scopuri diferite poate duce la programe greu de înţeles. Pentru a evita astfel de confuzii, asiguraţi-vă că folosiţi nume sugestive pentru fiecare variabilă în parte. În majoritatea cazurilor (cu excepţia variabilelor contor sau index), folosirea unor nume sugestive va elimina multe conflicte de nume. În al doilea rând, evitaţi pe cât posibil folosirea variabilelor globale. În sfârşit, dacă asemenea conflicte apar totuşi, asiguraţi-vă că aţi înţeles bine regulile de domeniu din C++, ca şi durata de viaţă a variabilelor, ce va fi discutată în continuare.

SĂ ÎNŢELEGEM DURATA DE VIAŢĂ A UNUI IDENTIFICATOR

Durata de viaţă a unei variabile defineşte perioada de timp în care variabila rămâne în domeniu. În mod normal, viaţa unei variabile începe când aceasta este definită şi se termină când iese din domeniu. De exemplu, când într-o funcţie se declară variabile locale, acestea intră în acţiune la apelarea funcţiei şi

îşi încheie existenţa la terminarea funcţiei. Analog, dacă un program defineşte o variabilă globală, durata de viaţă a acesteia începe la definiţie şi continuă până la terminarea programului.

După discuţia prezentată în acest capitol, se poate crede că durata de viaţă a unei variabile este o chesiune simplă. Din păcate, folosirea calificatorilor, cum este *static*, poate complica conceptul duratei de viaţă. De exemplu, următoarea funcţie, *use_count*, foloseşte variabila statică *counter* pentru a calcula numărul de apeluri ale funcţiei:

```
long use_count(void)
 {
   static long counter = 0;

   // Statements

   return(counter);
 }
```

În acest exemplu, durata de viaţă a variabilei *counter* începe la apelarea funcţiei, care la rândul său va iniţializa şi variabila. Spre deosebire de alte variabile locale declarate în interiorul funcţiei, a căror durată de viaţă se încheie o dată cu terminarea funcţiei, durata de viaţă a variabilei *counter* va continua până la terminarea programului.

Aşa cum aţi învăţat, C++ permite declararea variabilelor în orice parte a programului. Următoarea instrucţiune *if*, de exemplu, declară o variabilă contor statică.

```
if (some_condition)
 {
   static int if_counter = 0;

   // Statements

   if_counter++;
 }
```

Pentru a îmbunătăţi performanţa programului, se poate folosi o astfel de variabilă pentru a determina de câte ori se execută o anumită parte a codului. În acest caz, durata de viaţă a variabilei începe o dată cu prima execuţie a instrucţiunii *if* şi se încheie la terminarea programului.

Un obicei frecvent folosit în programele C++ este de a declara o variabilă în interiorul unui ciclu *for*, ca mai jos:

```
for (int i = 0; i < 10; i++)
    some_operation();
```

Când se declară o variabilă în acest mod, durata de viaţă a variabilei începe când ciclul este executat pentru prima dată şi se încheie când blocul care conţine ciclul este încheiat. Următoarea secvenţă defineşte variabilele *i* şi *j* în cadrul a două cicluri *for* incluse:

```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    cout << i << ' ' << j << endl;
```

La execuţia acestui fragment de program, ecranul va afişa:

```
01
02
03
10
11
12
20
21
22
```

După cum se observă, programul tratează corect variabilele *i* şi *j*. La prima vedere, se poate crede că secvenţa de cod creează variabilele *i* şi *j*, care vor continua să existe până la încheierea blocului de program în care sunt scrise cele două cicluri. Dar, aşa cum se va vedea, variabila *i* există până la sfârşitul blocului curent, în timp ce variabila *j* îşi încetează existenţa la terminarea ciclului *for* exterior. Următorul program, SYNTAX.CPP, încearcă să afişeze valorile ambelor variabile după încheierea ciclurilor. Dacă încercaţi să compilaţi acest program, compilatorul va genera o eroare de sintaxă, semnalând faptul că variabila *j* este necunoscută:

```
#include <iostream.h>

void main(void)
 {
   for (int i = 0; i < 3; i++)
     for (int j = 0; j < 3; j++)
       cout << i << ' ' << j << endl;      Identificator j necunoscut

   cout << "Ending values i: " << i << " j: " << j << endl;
 }
```

Când lucraţi cu mai multe variabile, încercaţi să determinaţi durata de viaţă a fiecăreia. Înţelegând mai bine domeniul şi durata de viaţă a unei variabile, veţi înţelege mai bine operaţiile de fineţe pe care le realizează compilatorul, având astfel posibilitatea de a scrie un cod mai eficient.

SĂ ÎNŢELEGEM EDITAREA LEGĂTURILOR

La programele mici, cum sunt cele prezentate în această carte, de obicei tot codul este inclus într-un singur fişier sursă. Pe măsură ce programele devin mai mari, veţi înţelege mai uşor codul programului dacă acesta este împărţit logic în mai multe fişiere. În plus, se poate reduce timpul de dezvoltare a programului prin compilarea separată a componentelor acestuia. În acest mod, dacă se

efectuează schimbări asupra unei anumite părţi a programului, va trebui recompilat un singur fişier sursă.

În continuare, se poate folosi editorul de legături pentru a *lega* fişierele obiect compilate anterior. În Capitolul 17 veţi învăţa cum să construiţi o bibliotecă de clase. Deocamdată, însă, veţi compila şi apoi lega două fişiere sursă înrudite. Să creăm mai întâi fişierul STR_LEN.CPP ce conţine următoarea funcţie:

```
int str_len(char *string)
 {
   int length = 0;

   while (*string++)
     length++;

   return(length);
 }
```

Prin compilarea acestui fişier se creează fişierul cu cod obiect STR_LEN.OBJ. Dacă se foloseşte compilatorul Borland C++, este necesară următoarea comandă:

```
C:\> BCC -c STR_LEN.CPP  <ENTER>
```

Observaţie: *Opţiunea "-c" din linia de comandă indică compilatorului să efectueze o compilare, fără a executa şi legarea. Fără această opţiune pe linia de comandă compilatorul va încerca să creeze fişierul STR_LEN.EXE (litera din opţiunea "-c" trebuie să fie literă mică).*

Apoi se creează fişierul HOWLONG.CPP care foloseşte funcţia anterioară:

```
#include <iostream.h>

int str_len(char *);

void main(void)
 {
   char *string = "Success with C++";

   cout << "The string " << string << " contains " <<
           str_len(string) << " characters" << endl;
 }
```

Prin compilarea acestui program se obţine fişierul obiect HOWLONG.OBJ:

```
C:\> BCC -c HOWLONG.CPP  <ENTER>
```

Cele două fişiere obiect se pot combina cu ajutorul editorului de legături, formându-se programul executabil HOWLONG.EXE. Dacă folosiţi Borland C++, se poate realiza legarea celor două fişiere cu ajutorul comenzii:

```
C:\> BCC  HOWLONG.OBJ  STR_LEN.OBJ  <ENTER>
```

Observaţie: *Multe compilatoare, printre care şi Borland C++, permit specificarea în linia de comandă a fişierelor obiect ce urmează a fi compilate şi apoi legate. Astfel, dacă folosiţi Borland C++ se poate compila programul HOWLONG.CPP şi apoi lega fişierul STR_LEN.OBJ folosind următoarea comandă:*

```
C:\> BCC HOWLONG.CPP STR_LEN.OBJ <ENTER>
```

Dacă examinaţi programul HOWLONG.CPP, veţi vedea că el include prototipul funcţiei *str_len*. Chiar dacă funcţia *str_len* se află în alt fişier, ea trebuie totuşi declarată înainte de a fi folosită.

Când programul foloseşte mai multe fişiere sursă sau obiect, trebuie înţelese *legăturile* (referinţele între entităţile programului) care sunt folosite de compilator şi de editorul de legături pentru a rezolva referinţele între mărimile programului. C++ defineşte două tipuri de legături, interne şi externe. Când un nume de identificator este local unui fişier sursă, identificatorul are *legătură internă*. Dacă domeniul unui identificator depăşeşte fişierul în care este declarat, identificatorul are *legătură externă*. Să presupunem, de exemplu, că un fişier sursă defineşte variabila globală *error_number* ca mai jos:

```
hms drrnr_mtladr:  .. Glnaal

unhc lahm(unhc(
 z
   .. Ssasdldmsr
 }
```

În acest caz, domeniul variabilei globale depăşeşte fişierul sursă, prin urmare variabila are legătură externă. De exemplu, să creăm următorul program ESSOSNUM.CPP:

```
unhc rgnv_drrnr_mtladr(unhc(:

hms drrnr_mtladr = 1001:

unhc lahm(unhc(
 z
   rgnv_drrnr_mtladr((:
 }
```

După cum se poate vedea, programul atribuie o valoare variabilei globale şi apoi apelează funcţia *show_error_number* pentru a afişa valoarea variabilei. Ca şi înainte, programul include un prototip pentru funcţie, chiar dacă funcţia este declarată în alt fişier.

Să creăm acum fişierul SHOWESS.CPP, care defineşte funcţia *show_error_number*:

```
#hmb1tcd <hnrsrdal-g>

dxsdrm hms drrnr_mtladr:

unhc rgnv_drrnr_mtladr(unhc(
 z
```



```
   cout << "Error number: " << error_number << endl;
 }
```

În acest caz, fişierul foloseşte declaraţia *extern* pentru a informa compilatorul că variabila *error_number* este definită în alt fişier. După compilare, editorul de legături trebuie să rezolve referinţa la variabilă.

Când se scriu programe multifişier ce se bazează pe astfel de legături exerne, trebuie specificat corect tipul variabilei externe. În majoritatea cazurilor, compilatorul nu poate detecta o variabilă externă de tip greşit declarat, ceea ce duce la o eroare foarte dificil de detectat. De exemplu, următorul fişier, WRONGERR.CPP declară în mod incorect tipul variabilei externe *error_number* ca fiind *float*:

```
#include <iostream.h>

extern float error_number; ———— Specificare de tip incorectă

void show_error_number(void)
 {
   cout << "Error number: " << error_number << endl;
 }
```

Următorul fişier, BAD_INIT.CPP, atribuie o valoare iniţială variabilei externe *error_number*. Din păcate, atunci când compilatorul realizează iniţializarea, declaraţia devine o definiţie, ascunzând variabila globală externă anterioară:

```
#include <iostream.h>

extern int error_number = 2002;

void show_error_number(void)
 {
   cout << "Error number: " << error_number << endl;
 }
```

Dacă se compilează şi se leagă programele BAD_INIT.CPP şi ERRORNUM.CPP, se va afişa pe ecran următorul mesaj:

```
C:\> ERRORNUM <ENTER>
Error number: 2002
```

În funcţie de editorul de legături folosit, identificatorul dublu definit poate fi detectat. În caz contrar, depanarea erorii este o sarcină foarte dificilă. Pentru a înţelege mai bine prelucrarea realizată, s-ar putea modifica programul ERRORNUM.CPP pentru a afişa valoare a lui *error_number* astfel:

```
#include <iostream.h>

void show_error_number(void);
```

```
int error_number = 1001;

void main(void)
 {
   show_error_number();
   cout << "In main: " << error_number << endl;
 }
```

În acest caz, ieşirea programului devine:

```
C:\> ERRORNUM <ENTER>
Error number: 2002
In main 1001
```

Aşa cum se folosesc membrii de tip *private* ai unei clase pentru a nu putea fi apelaţi din afara obiectului, se poate limita domeniul unei variabile sau funcţii la fişierul sursă curent. Dacă în faţa numelui unei variabile globale se scrie cuvântul cheie *static*, atunci legarea acelei variabile devine *internă* fişierului sursă. Cu alte cuvinte, variabila nu mai poate fi utilizată în afara fişierului sursă curent. Următorul program, INTERNAL.CPP, defineşte două funcţii denumite *internal* şi *external*. Deoarece funcţia *internal* este precedată de cuvântul *static*, domeniul funcţiei este local fişierului sursă, pe când funcţia *external* are legătură externă:

```
#include <iostream.h>

static void internal(char *message)
 {
   cout << message << endl;
 }

void external(char *message)
 {
   cout << message << endl;
 }

void some_function(void);

void main(void)
 {
   internal("Using the internal function");
   external("Using the external function");

   some_function();
 }
```

Să observăm că în program se foloseşte şi funcţia *some_function*, care este declarată în afara fişierului sursă. Următorul program, SOMEFUNC.CPP defineşte *some_function*:

```
void external(char *);

void some_function(void)
 {
   external("Using external from outside the source file");
 }
```


După cum se vede, funcţia *some_function* apelează funcţia *external* pentru a afişa un mesaj. După compilarea şi legarea celor două fişiere, se lansează în execuţii programul INTERNAL care va tipări:

```
C:\> INTERNAL <Enter>
Using the internal function
Using the external function
Using external from outside the source file
```

Dacă se modifică fişierul SOMEFUNC.CPP pentru a folosi funcţia *internal*, editorul de legături va afişa un mesaj de eroare referitor la nerezolvarea referinţei la funcţia *internal*. Deoarece funcţia *internal* are o legătură internă, ea nu poate fi folosită în afara fişierului sursă.

La proiectarea programelor multifişier şi a bibliotecilor de programe, trebuie avută în vedere legarea referinţelor variabilelor şi funcţiilor. Aşa cum aţi văzut, erorile din timpul legării referinţelor pot fi la fel de greu de detectat, ca şi erorile de domeniu ale entităţilor programului.

REZUMAT

Acest capitol a analizat regulile domeniului de vizibilitate a mărimilor programului. Dacă încă nu vă este totul clar, aveţi răbdare şi exersaţi cu programele prezentate, observând influenţa modificărilor efectuate. În Capitolul 10 veţi examina referinţele C++, care oferă un nume alternativ pentru o variabilă. Înainte de a continua cu Capitolul 10, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Domeniul unui identificator defineşte partea din program în care identificatorul poate fi folosit.
- ✓ Înainte de folosire, un identificator trebuie declarat. În general, o declaraţie specifică un tip şi un nume. Dacă la declaraţii se alocă şi memorie, atunci declaraţia se consideră o definiţie.
- ✓ C++ acceptă 4 tipuri de domenii: local, de funcţie, de fişier, de clasă.
- ✓ Domeniul local corespunde mărimilor definite într-un bloc (între acolade).
- ✓ Domeniul de funcţii corespunde numai etichetelor de salt. O instrucţiune *goto* nu poate efectua saltul la o etichetă în afara funcţiei curente (*goto* este permis numai local).
- ✓ Domeniul de fişier corespunde variabilelor globale şi identificatorilor de funcţii care sunt cunoscuţi în tot fişierul sursă (sau în mai multe fişiere).
- ✓ Domeniul de clasă corespunde variabilelor membru şi funcţiilor membru care sunt accesibile numai obiectelor clasei.

✓ C++ permite declararea variabilelor intercalată cu instrucţiunile programului, nu numai la începutul unui bloc. În multe cazuri, declararea unei variabile aproape de locul de folosire îmbunătăţeşte claritatea programului.

✓ Dacă numele unei variabile locale intră în conflict cu cel al unei variabile globale sau al unei variabile membru a unei clase, compilatorul va asocia referinţele la acel nume cu variabila locală. Pentru a accesa variabila globală sau variabila clasei, trebuie folosit operatorul de rezoluţie globală (::).

✓ Durata de viaţă a unei variabile defineşte perioada în timpul căreia variabila rămâne în domeniu. În mod normal, existenţa unei variabile începe în momentul definirii ei.

✓ C++ acceptă legături interne şi externe ale entităţilor unui program. Identificatorii cu legătură internă sunt cunoscuţi numai în fişierul sursă în care sunt definiţi. Identificatorii cu legătură externă pot fi apelaţi din diferite fişiere sursă.

CAPITOLUL 10
ACOMODAREA CU REFERINŢELE C++

Dacă sunteţi un programator C experimentat, probabil că vă descurcaţi fără probleme cu operaţiile cu pointeri. De aceea ar trebui să fiţi pregătiţi pentru a înţelege *referinţele C++*, care sunt un fel de adrese, dar şi un fel de valori. Acest capitol examinează referinţele C++ în detaliu. Totuşi, în majoritatea cazurilor, veţi continua să folosiţi operaţiile standard cu pointeri, aşa cum aţi făcut şi până acum.

Programatorii sunt familiari cu majoritatea operaţiilor cu pointeri. Însă apar situaţii când lucraţi cu variabile complexe, cum sunt structurile şi obiectele, când folosirea referinţelor este foarte eficientă. În plus, în Capitolul 11 veţi învăţa să creaţi proprii manipulatori I/O, cum sunt *hex* sau *endl*. Aşa cum veţi vedea, manipulatorii lucrează cu referinţe la anumite stream-uri I/O.

La terminarea acestui capitol, veţi cunoaşte:

- Ce este o referinţă
- Capcanele numelor asociate
- Ce trebuie făcut pentru a crea o referinţă
- Ce operaţii se pot efectua cu referinţe
- Cum vede compilatorul C++ o referinţă
- Folosirea referinţelor
- Ce sunt obiectele ascunse

Dacă se cunosc funcţiile care primesc şi returnează referinţe, manipulatorii de stream I/O vor fi uşor de creat.

O REFERINŢĂ ESTE UN NUME ASOCIAT

Aşa cum cunoaşteţi, un nume de variabilă corespunde unei locaţii de memorie ce este accesibilă folosind o anumită adresă. O *referinţă* C++ este un al doilea nume (sau *alias*) pentru o locaţie de memorie. Înainte de a folosi o referinţă, aceasta trebuie declarată în program. Ca şi alte declaraţii, o referinţă are un

nume şi un tip. Când declaraţi o referinţă, trebuie să atribuiţi referinţei variabila pentru care referinţa serveşte drept nume asociat (*alias*). Declaraţiile de referinţe plasează caracterul ampersand (&) imediat după tipul variabilei, ca de exemplu *int&*. Următoarele instrucţiuni, de exemplu, declară variabila *alias* ca o referinţă la variabila *count*:

```
int count;

int&  alias = count;    // Create the reference
```

Variabilă la care referinţa este alias

Nume referinţă (alias)

Referinţă de tip **int**

Următorul program, FIRSTREF.CPP, creează o referinţă denumită *alias* pentru variabila *count*. Programul afişează valoarea lui *count* folosind atât variabila, cât şi referinţa. Programul incrementează apoi valoarea lui *count* incrementând variabila *alias*:

```
#include <iostream.h>

void main(void)
 {
   int count = 1000;

   int& alias = count;  // Create the reference

   cout << "count's value is " << count << " and " << alias << endl;

   alias++;             // Increment count's value

   cout << "count's value is " << count << " and " << alias << endl;
 }
```

O referinţă nu este altceva decât un al doilea nume (alias) pentru o variabilă. După crearea unei referinţe, programul poate folosi fie referinţa, fie variabila, pentru a avea acces la aceeaşi locaţie de memorie. În exemplul precedent, programul nu numai a afişat valoarea lui *count* folosind referinţa, dar a folosit referinţa şi pentru incrementarea valorii acesteia.

La compilarea şi execuţia acestui program pe ecran va apărea:

```
C:\> FIRSTREF  <ENTER>
count's value is 1000 and 1000
count's value is 1001 and 1001
```

Se pot crea referinţe pentru variabile de orice tip (*float*, *long* sau chiar *struct*). Următorul program, REFTYPES.CPP, de exemplu, creează referinţe pentru diferite tipuri de date şi le foloseşte pentru manevrarea valorilor variabilelor asociate:


```
#include <iostream.h>

void main(void)
 {
   float pi = 22.0 / 7.0;

   float& pi_alias = pi;

   struct Date { int month; int day; int year; }
               today = { 9, 30, 94 };

   Date& birthday = today;

   cout << "Pi is " << pi << " or " << pi_alias << endl;

   cout << "My birthday is " << birthday.month << '/' <<
           birthday.day << '/' << birthday.year << endl;
 }
```

După cum se observă, programul creează două referinţe, una de tipul *float* şi alta de tipul *Date*. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> REFTYPES  <ENTER>
Pi is 3.142857 or 3.142857
My birthday is 9/30/94
```

Scopul acestor două prime programe este acela de a arăta că o referinţă este pur şi simplu un alias sau al doilea nume al unei variabile. În mod normal, referinţele nu se folosesc în maniera indicată aici. Folosind două sau mai multe nume pentru aceeaşi locaţie de memorie, programele devin mai greu de înţeles. Cel mai frecvent, referinţele se folosesc pentru transferul de valori către funcţii şi dinspre funcţii, în special pentru valorile complexe de tip structuri sau obiecte.



CREAREA UNEI REFERINŢE

O referinţă este un nume asociat (sau alias) pentru o variabilă. Referinţele au nume şi tip (ca *int*, *float*, *class* sau *struct*). Pentru a crea o referinţă, ea trebuie declarată, similar cu modul în care declarăm variabila. De exemplu, următoarele instrucţiuni creează o referinţă denumită *hillary* pentru o variabilă de tipul *president* denumită *bill*:

```
struct president { char name[64]; int age; short qualifications;
                 } bill;

president& hillary = bill;
```

Spre deosebire de o variabilă, totuşi, valoarea atribuită unei referinţe nu poate fi schimbată. O dată ce aţi atribuit o valoare unei referinţe, această valoare rămâne asociată pentru toată durata execuţiei blocului de program.

O REFERINȚĂ NU ESTE O VARIABILĂ

Deşi o referinţă seamănă cu o variabilă, în sensul că trebuie specificate un nume şi un tip şi trebuie atribuită o valoare, o referinţă nu este o variabilă. O dată ce aţi atribuit o valoare unei referinţe, referinţa nu se poate schimba. Cu alte cuvinte, o dată ce aţi asociat un nume unei variabile, nu puteţi folosi acelaşi nume şi pentru o altă variabilă.

Următorul program REF_FUNC.CPP, foloseşte o variabilă referinţă la un parametru al funcţiei. Când se foloseşte o referinţă locală în acest mod, ea rămâne validă pe durata funcţiei. De fiecare dată când se apelează funcţia, referinţa va corespunde noului parametru transmis. Funcţia nu poate modifica valoarea referinţei:

```
#include <iostream.h>

void use_reference(int value)
 {
   int& alias = value;

   cout << "The parameter value using alias is " << alias << endl;
 }

void main(void)
 {
   for (int i = 0; i < 5; i++)
     use_reference(i);
 }
```

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> REF_FUNC  <ENTER>
The parameter value using alias is 0
The parameter value using alias is 1
The parameter value using alias is 2
The parameter value using alias is 3
The parameter value using alias is 4
```

Aşa cum se observă, referinţa este asignată noului parametru de fiecare dată când funcţia este invocată.



POSIBILE SINTAXE PENTRU REFERINŢE

Toate referinţele din programele prezentate în această carte plasează ampersand-ul (&) imediat după numele tipului, ca mai jos:

```
int value;

int& alias = value;
```

C++ vă permite, totuşi, să plasaţi ampersand-ul între numele tipului şi numele referinţei, sau chiar imediat după numele referinţei. Următoarele declaraţii, prin urmare, sunt identice:

```
int& alias = value;      int & alias = value;      int &alias = value;
```

Cu toate că puteţi întâlni toate cele 3 formate anterioare, este bine să alegeţi unul din ele şi să îl folosiţi constant. În acest mod programele vor deveni mai consecvente (uniforme) în notaţii, şi deci mai uşor de înţeles.

REGULI PENTRU LUCRUL CU REFERINŢE

Aşa cum aţi învăţat, o referinţă creează un al doilea nume pentru o variabilă. Multe din operaţiile care se pot efectua cu variabila, se pot efectua şi cu referinţa ei. De exemplu, se pot efectua operaţii aritmetice, de atribuire, se poate afişa valoarea variabilei, se poate chiar testa valoarea în operatori condiţionali. Rămân, însă, o serie de operaţii care nu pot fi efectuate pe referinţa unei variabile.

Programatorul nu are capacitatea de a:

- obţine adresa unei referinţe folosind operatorul de adresă (&)
- atribui un pointer unei referinţe
- compara două valori referinţă
- crea deplasamentul (offset) unei referinţe
- modifica valoarea unei referinţe

Să nu înţelegeţi greşit regula despre folosirea unei referinţe într-o operaţie condiţională. De exemplu, următorul program, TESTREF.CPP, foloseşte referinţa *alias* într-o instrucţiune *if* pentru a testa dacă valoarea din locaţia de memorie referită de *alias* este 1001:

```
#include <iostream.h>

void main(void)
 {
   int value = 1001;

   int& alias = value;

   if (alias == 1001)
     cout << "alias contains the value 1001" << endl;
   else
     cout << "alias does not the value 1001" << endl;
 }
```

După cum se observă, programul foloseşte referinţa într-un test condiţional. Dar un asemenea test nu se poate folosi pentru a determina dacă două referinţe adresează aceeaşi locaţie de memorie. În schimb, se pot compara valorile din fiecare adresă de memorie corespunzătoare referinţelor. De exemplu, următorul program, TEST2REF.CPP, foloseşte o instrucţiune *if* pentru a compara valorile referite de *alias_one* şi *alias_two*:

```
#include <iostream.h>

void main(void)
 {
   int one = 1;
   int& alias_one = one;

   int uno = 1;
   int& alias_two = uno;

   if (alias_one == alias_one)
     cout << "The references contain the same value " <<
             alias_one << endl;
   else
     cout << "The alias values differ " << alias_one << " and " <<
             alias_two << endl;
 }
```

Aşa cum se observă, instrucţiunea *if* compară valorile celor două referinţe. Dacă compilaţi şi executaţi acest program, pe ecran va apărea:

```
C:\> TEST2REF  <ENTER>
The references contain the same value 1
```

Se observă că instrucţiunea *if* testează valorile conţinute în locaţiile de memorie asociate referinţelor, şi nu adresele de memorie corespunzătoare fiecărei referinţe. În acest exemplu, cele două referinţe nu sunt dependente. Întâmplător, locaţiile de memorie corespunzătoare referinţelor conţin valoarea 1.

REFERINŢA ESTE O VALOARE SAU O ADRESĂ?

Mulţi programatori începători în lucrul cu referinţele încearcă să le identifice cu adresele. Cea mai bună imagine a unei referinţe este o *adresă rezolvată*. Pentru compilatorul C++, o referinţă este o adresă. Când compilatorul generează codul obiect, referinţele sunt tratate drept adrese. În cadrul programului, însă, referinţa corespunde unei valori ce se află la o anumită adresă. Cu alte cuvinte, o referinţă permite programului să manipuleze o valoare dintr-o locaţie de memorie, fără a fi nevoie de a lucra cu pointeri. Orice operaţie care poate fi efectuată cu referinţe poate fi efectuată şi cu pointeri.

Referinţele sunt uşor de folosit în cadrul programului, deoarece compilatorul este cel care rezolvă, în fundal, problemele legate de adrese şi valori. Întrucât referinţele nu sunt nici variabile, nici pointeri, există anumite operaţii pe care acestea nu le acceptă.

O modalitate de a obţine o imagine reală asupra modului de manevrare a referinţelor de către compilator este de a genera listingul în limbaj de asamblare al programului. Dacă folosiţi compilatorul Borland C++, un astfel de listing se obţine folosind comutatorul -S, ca mai jos:

```
C:\> BCC -S FILENAME.CPP <ENTER>
```

Sintaxa pentru compilatorul Microsoft Visual C++ este următoarea:

```
C:\> CL /Fa /Fs FILENAME.CPP <ENTER>
```

Observaţie: *La ambele compilatoare, comutatorii din linia de comandă sunt sensibili la diferenţa dintre majuscule şi literele mici.*

După ce obţineţi listingul în limbaj de asamblare, modificaţi conţinutul fişierului şi observaţi cu atenţie instrucţiunile folosite de compilator pentru rezolvarea adreselor.

FOLOSIREA REFERINŢELOR DREPT PARAMETRI

Referinţele dau posibilitate programelor să beneficieze de avantajele pointerilor, fără o sintaxă complicată (cu alte cuvinte, nu trebuie transferate adresele parametrilor şi apoi folosiţi pointerii pentru a obţine valorile acestora). Următorul program, REFPARAM.CPP, de exemplu, transferă o referinţă funcţiei *change_value*. În cadrul funcţiei, se foloseşte un parametru de tip referinţă pentru a se putea schimba valoarea variabilei asociată acestui parametru:

```
#include <iostream.h>

void change_value(int& value)          Parametru referinţă
 {
```

```
   value = 1001;
 }

void main(void)
 {
   int count = 1;

   int& alias = count;

   cout << "count's value is " << count << " or " << alias << endl;

   change_value(alias);          Transferul referinţei ca parametru

   cout << "count's value is " << count << " or " << alias << endl;
 }
```

Se observă că programul transferă referinţa *alias* funcţiei *change_value*. Funcţia foloseşte o altă referinţă, denumită *value*. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> REFPARAM  <ENTER>
count's value is 1 or 1
count's value is 1001 or 1001
```

Este evident că programul elimină nevoia de a lucra cu pointeri şi adrese. Totuşi, programul introduce un nume suplimentar (în acest caz *alias*) pe care programatorul trebuie să-l reţină. De regulă, cu cât sunt mai puţine nume de variabile şi referinţe de memorat de către utilizator, cu atât programul va fi mai uşor de citit şi modificat.

Folosind pointeri, se poate obţine acelaşi rezultat, cum se arată în programul următor, PTRPARAM.CPP:

```
#include <iostream.h>

void change_value(int *value)
 {
   *value = 1001;              Argument pointer
 }
void main(void)
 {
   int count = 1;

   cout << "count's value is " << count << endl;

   change_value(&count);        Transferul unui argument prin adresă

   cout << "count's value is " << count << endl;
 }
```


În mod similar, următorul program, REF_EXCH.CPP, transferă două referinţe funcţiei *exchange_values*, care va schimba între ele valorile acestora:

```
#include <iostream.h>

void exchange_values(int &a, int& b)
 {
   int temp;     // Not a reference        Doi parametri referinţă

   temp = a;
   a = b;
   b = temp;
 }

void main(void)
 {
   int one = 1;
   int two = 2;

   int& one_alias = one;
   int& two_alias = two;

   cout << "Before exchange: one is " << one << " and two is " <<
           two << endl;

   exchange_values(one_alias, two_alias);

                               Transferul referinţelor ca parametri

   cout << "After exchange: one is " << one << " and two is " <<
           two << endl;
 }
```

Ca şi mai înainte, programul elimină nevoia de a lucra cu pointeri şi adrese, dar adaugă două referinţe (*one_alias* şi *two_alias*). Când numărul variabilelor şi referinţelor folosite într-un program creşte, va creşte şi gradul de dificultate al acestuia. Să observăm că, în cadrul funcţiei *exchange_values*, variabila *temp* este declarată de tip *int*, şi nu referinţă la tipul *int*. Reţineţi că instrucţiunea ce urmează declaraţiei variabilei *temp* atribuie acesteia valoarea conţinută în locaţia de memorie asociată variabilei *a* (care este 1). Dacă funcţia ar fi declarat *temp* ca o referinţă, atunci *temp* ar fi devenit un al doilea nume pentru variabila *a*:

```
int &temp = a;     // Create an alias
```

Exersaţi acest program pentru a vă asigura că aţi înţeles de ce variabila *temp* nu trebuie să fie referinţă. Următorul program, PTR_EXCH.CPP, realizează acelaşi schimb de valori, dar folosind pointeri. Din acelaşi motiv, variabila *temp* este declarată de tip *int*, şi nu pointer la *int*:

```
#include <iostream.h>

void exchange_values(int *a, int *b)
 {
   int temp;    // Not a pointer

   temp = *a;
   *a = *b;
   *b = temp;
 }

void main(void)
 {
   int one = 1;
   int two = 2;

   cout << "Before exchange: one is " << one << " and two is " <<
           two << endl;

   exchange_values(&one, &two);

   cout << "After exchange: one is " << one << " and two is " <<
           two << endl;
 }
```

FOLOSIREA REFERINŢELOR CU PARAMETRII ŞI OBIECTELE UNEI STRUCTURI

O utilizare frecventă a referinţelor este de a transfera parametri de structuri către funcţii sau de a returna variabile complexe. De exemplu, următorul program, SHOWDATE.CPP, foloseşte funcţiile *get_date* şi *show_date* pentru a cere utilizatorului data curentă şi apoi pentru a o afişa. Programul memorează data într-o structură de tip *Date*, ca mai jos:

```
struct Date {
  int month;
  int day;
  int year;
};
```

Veţi vedea că programul transferă o referinţă la o variabilă de tip *Date* funcţiei *get_date* şi chiar variabila de tipul *Date* funcţiei *show_date*:

```
#include <iostream.h>

struct Date {
 int month;
 int day;
```



```
 int year;
};

void get_date(Date& date)
 {
   cout << "Type in today's month: ";
   cin >> date.month;

   cout << "Type in today's day: ";
   cin >> date.day;

   cout << "Type in today's year: ";
   cin >> date.year;
 }

void show_date(Date date)
 {
   cout << "Today's date is: " << date.month << '/' << date.day <<
           '/' << date.year << endl;
 }

void main(void)
 {
   Date today;

   Date& todays_date = today;

   get_date(todays_date);

   show_date(today);
 }
```

Aşa cum se observă, programul transferă o referinţă funcţiei *get_date*, dar variabila de tip structură este transferată funcţiei *show_date*. Deoarece funcţia *get_date* modifică membrii structurii, ea trebuie să lucreze fie cu pointeri, fie cu referinţe la structură.

Folosind referinţe, funcţia elimină nevoia de a folosi operaţii cu pointeri, de felul următor:

```
void get_date(Date *date)
 {
   cout << "Type in today's month: ";
   cin >> date->month;

   cout << "Type in today's day: ";
   cin >> date->day;
```

```
  cout << "Type in today's year: ";
  cin >> date->year;
}
```



EXPLICAŢI OPERAŢIILE CU REFERINŢE

Majoritatea programatorilor C şi C++ sunt familiarizaţi cu operaţiile pe pointeri. Dacă alegeţi referinţele, explicaţi în detaliu aceste operaţii. Altfel, cei care citesc programul vor sesiza cu dificultate diferenţa dintre numele de variabile similare (variabila şi referinţa ei). În cazul programului SHOWDATE.CPP anterior, comentariile la funcţia *main* s-ar putea face astfel:

```
void main(void)
 {
   Date today;

   Date& todays_date = today;  // Create an alias for today that
                               // eliminates pointer operations
                               // Within functions that change
                               // structure members

   get_date(todays_date);      // Todays_date is a reference that
                               // aliases the today structure

   show_date(today);           // Pass the actual date structure--
                               // not the reference
 }
```

Când o funcţie returnează o dată de tip structură, aceasta se face cu un aport considerabil de memorie, deoarece membrii structurii trebuie copiaţi din stivă în noua structură, căreia i-au fost atribuiţi. Să presupunem, de exemplu, că funcţia *get_date* returnează o dată de tipul *Date*, cum se arată în programul RET_STRU.CPP:

```
#include <iostream.h>

struct Date {
  int month;
  int day;
  int year;
};             ——— Funcţia returnează o structură de tip Date

Date get_date(void)
 {
   Date date;
```



```
   cout << "Type in today's month: ";
   cin >> date.month;

   cout << "Type in today's day: ";
   cin >> date.day;

   cout << "Type in today's year: ";
   cin >> date.year;

   return(date);
 }

void show_date(Date date)
 {
   cout << "Today's date is: " << date.month << '/' << date.day <<
          '/' << date.year << endl;
 }

void main(void)
 {
   Date today;

   today = get_date(); ——— Atribuirea valorii returnate

   show_date(today);
 }
```

Dacă generaţi un listing în limbajul de asamblare pentru programul precedent, aţi putea determina modul în care compilatorul manipulează stiva pentru a transfera şi returna structurile.

Pentru a elimina surplusul de memorie impus de manipularea stivei, se poate folosi o referinţă la structură, cum se arată în următorul program, RET_REF.CPP:

```
#include <iostream.h>

struct Date {
  int month;
  int day;
  int year;
};

Date& get_date(Date &date)
 {
   cout << "Type in today's month: ";
   cin >> date.month;

   cout << "Type in today's day: ";
   cin >> date.day;
```

```
    cout << "Type in today's year: ";
    cin >> date.year;

    return(date);
 }

void show_date(Date& date)
 {
    cout << "Today's date is: " << date.month << '/' << date.day <<
          '/' << date.year << endl;
 }

void main(void)
 {
    Date today;

    Date& todays_date = today;  // Alias for today

    todays_date = get_date(todays_date);

    show_date(todays_date);
 }
```

Aşa cum se observă, ambele funcţii, *get_date* şi *show_date*, lucrează cu referinţe la structuri. În consecinţă, programul capătă un plus de consecvenţă. În plus, se poate modifica funcţia *main* prin combinarea a două instrucţiuni, astfel:

```
todays_date = get_date(todays_date);      show_date(get_date(todays_date));
show_date(todays_date);
```

Funcţia precedentă *get_date* primea o referinţă ca parametru şi apoi returna aceeaşi referinţă. Aşa cum veţi învăţa, mulţi dintre manipulatorii de stream I/O au fost concepuţi pentru o comportare asemănătoare. Manipulatorii primesc o referinţă ca parametru al stream-ului, realizează o operaţie, apoi returnează acea referinţă.

FOLOSIREA REFERINŢELOR LA OBIECTE

Dacă examinaţi diverse programe în C++, veţi întâlni funcţii care folosesc sau returnează referinţe la obiecte. De exemplu, în Capitolul 11, veţi învăţa cum se creează proprii manipulatori de stream-uri I/O, cum sunt *hex*, *dec* sau *oct*. Veţi defini atunci funcţii manipulator ce lucrează cu stream-uri I/O similare cu cele arătate în programul următor, DECOCTHE.CPP:

```
#include <iostream.h>
#include <iomanip.h>
                        ——— Returnează o referinţă
ios& dec(ios& stream)
 {                      ——— Parametru tip referinţă
```



```
   stream.setf(ios::dec);
   return(stream);
 }

ios& oct(ios& stream)
 {
   stream.setf(ios::oct);
   return(stream);
 }

ios& hex(ios& stream)
 {
   stream.setf(ios::hex);
   return(stream);
 }

void main(void)
 {
   cout << oct << "Value 10 in octal is " << 10 << endl;
   cout << dec << "Value 10 in decimal is " << 10 << endl;
   cout << hex << "Value 10 in hexadecimal is " << 10 << endl;
 }
```

Când manipulatorul apare într-o operaţie de ieşire, C++ va apela funcţia manipulator corespunzătoare. După cum se vede, funcţia primeşte ca parametru o referinţă la un stream I/O şi apoi returnează o referinţă la acelaşi stream:

```
ios& dec(ios& stream)
 {
   stream.setf(ios::dec);
   return(stream);
 }
```

Următoarea funcţie *tab*, de exemplu, inserează caracterul de tabulare într-un stream de ieşire:

```
ostream& tab(ostream& stream)
 {
   stream << '\t';
   return(stream);
 }
```

După cum se vede, manipulatorul inserează caracterul *tab* în stream-ul de ieşire specificat. Când operaţia s-a încheiat, funcţia va returna referinţa la stream. Pentru a folosi manipulatorul, programul îl va include în stream-ul de ieşire, cum se arată mai jos:

```
cout << tab << "Indenting the program output" << endl;
```

ATENȚIE LA OBIECTELE ASCUNSE

Când un program folosește referințe, trebuie să vă asigurați că tipul referinței corespunde cu cel al variabilei asociate. De exemplu, dacă se creează un alias la o variabilă de tipul *int*, trebuie să declarați referința de tip *int*:

```
cout << tab << "Indenting the program output" << endl;

int& alias = value;
```

Dacă tipul referinței nu concordă cu tipul variabilei, compilatorul va crea un *obiect ascuns*, căruia i se va atribui valoarea variabilei. De exemplu, următorul program, WRONGREF.CPP, creează o referință denumită *wrong_type* de tipul *long* la care atribuie un alias la o variabilă de tipul *int*:

```
#include <iostream.h>

void main(void)
 {
   int value = 1;

   long& wrong_type = value;

   cout << "Starting values: " << value << " and " << wrong_type <<
          endl;

   value++;

   cout << "Ending values: " << value << " and " << wrong_type <<
          endl;
 }
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> WRONGREF  <Enter>
Starting values: 1 and 1
Ending values: 2 and 1
```

După cum se poate vedea, variabila și referința sa conțin valori diferite! Când compilatorul C++ a întâlnit referința la tipul *long*, el a alocat memorie pentru o altă mărime, căreia i-a asociat referința *wrong_type*. După ce compilatorul a alocat o locație de memorie, el i-a atribuit acesteia valoarea variabilei pentru care a fost creat aliasul (adică valoarea 1). Exersați cu acest program, efectuând operații pe variabila *value* și pe referința *wrong_type*. Veți vedea că variabila și referința sunt independente.

Majoritatea programelor nu vor folosi niciodată obiecte ascunse. Din păcate, acestea pot duce la erori greu de depanat. Prin urmare, trebuie să fiți conștient de această caracteristică a compilatorului.

REZUMAT

La examinarea programelor C++, puteți întâlni funcții ce folosesc în mare măsură referințele. În cel mai simplu sens, o referință este un alias sau un al doilea nume folosit pentru o variabilă. În Capitolul 11, de exemplu, veți crea funcții manipulator ce primesc ca parametru o referință la un stream I/O și returnează apoi tot o referință la stream. Înainte de a continua cu Capitolul 11, asigurați-vă că ați învățat următoarele:

- ✓ O referință este un alias sau un al doilea nume folosit de program pentru a menționa o variabilă.
- ✓ Folosind unul sau mai multe nume asociate pentru aceeași variabilă, programele pot deveni mai greu de înțeles, prin creșterea numărului de identificatori pe care programatorul trebuie să-i rețină.
- ✓ Pentru a crea o referință, trebuie specificat un nume și un tip, plasând simbolul ampersand (&) imediat după numele tipului. În plus, e necesar ca referinței să i se atribuie o valoare inițială.
- ✓ O referință nu este o variabilă. De aceea, C++ restrânge setul de operații ce pot fi efectuate cu o referință.
- ✓ Pentru compilatorul C++, o referință este o adresă. Pentru program, o referință este văzută ca o valoare conținută la o adresă rezolvată.
- ✓ Prin folosirea referințelor, se pot elimina multe din operațiile cu pointeri. De aceea, multe programe C++ folosesc referințele pentru a transfera sau a returna structuri și obiecte către sau dinspre o funcție.
- ✓ Orice operație efectuată cu o referință poate fi efectuată și cu pointeri.
- ✓ Dacă tipul referinței nu coincide cu tipul variabilei asociate, compilatorul C++ va crea un obiect ascuns. Astfel de obiecte sunt dificil de detectat.

SUCCES CU C++

SECȚIUNEA II

APROFUNDAREA



CAPITOLUL 11
APROFUNDAREA STREAM-URILOR I/O DIN C++

În Capitolul 1 aţi examinat stream-urile I/O din C++ şi modul lor de folosire. Atunci aţi învăţat să folosiţi diferiţi manipulatori şi funcţii membru. Acest capitol analizează mai în amănunt clasele care formează stream-urile I/O din C++. Veţi examina în detaliu clasele de bază şi clasele derivate. În momentul încheierii acestui capitol veţi putea înţelege pe deplin fişierele antet, cum sunt IOSTREAM.H şi IOMANIP.H. În plus, veţi învăţa să creaţi propriii dumneavoastră manipulatori. În sfârşit, acest capitol examinează stream-ul de date ce are loc în fundal când este executată o operaţie de intrare/ieşire:

La terminarea acestui capitol, veţi cunoaşte următoarele:

- Cum se pot crea manipulatori cu parametri
- Cum se poate simplifica folosirea unui manipulator în alte programe
- Modul în care se pot asocia două stream-uri şi efectul acestei operaţii
- Unde sunt definite stream-urile de fişier în C++
- Cum se pot realiza operaţii I/O formatate
- Cum se realizează operaţii I/O neformatate

Observaţie: *Compilatorul pe care îl aveţi la dispoziţie poate folosi şi alte fişiere antet, decât cele descrise în acest capitol. De exemplu, compilatorul Borland C++ versiunea 4.0 foloseşte fişierul IOSTREAM.H pentru toate definiţiile legate de stream-uri. În plus, faţă de IOSTREAM.H, compilatorul Microsoft Visual C++ versiunea 1.0 foloseşte şi fişierele IOS.H, ISTREAM.H şi OSTREAM.H. Acest capitol foloseşte Borland C++ ca punct de referinţă; dacă folosiţi alte compilatoare, va trebui să utilizaţi fişierele antet impuse de acestea.*

SĂ ÎNŢELEGEM RELAŢIA ÎNTRE CLASELE STREAM-URILOR I/O

Aşa cum aţi învăţat, fişierul antet IOSTREAM.H defineşte clasele pentru stream-uri I/O. Dacă nu aţi făcut-o încă, tipăriţi acum o copie a acestui fişier. Când examinaţi conţinutul acestui fişier, veţi găsi clase de genul *ios*, *istream* şi *ostream*. După cum se vede în Figura 11.1, clasa *ios* este clasa de bază, din care se derivează clasele *istream* şi *ostream*. Clasele *istream* şi *ostream* adaugă la clasa de bază *ios* operatorii de *extracţie* (>>), respectiv de *inserţie* (<<).

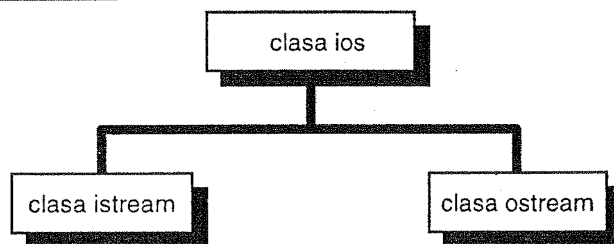


*Figura 11.1 Clasele **istream** şi **ostream** sunt derivate din clasa de bază **ios**.*

Clasele *istream* şi *ostream* implementează operatorii de extracţie şi inserţie. Dacă examinaţi fişierul antet IOSTREAM.H, veţi găsi prototipuri pentru operatorii de inserţie şi extracţie, de forma:

```
istream _FAR & _Cdecl operator>> (istream _FAR & (_Cdecl *_f)(istream _FAR
&));
istream _FAR & _Cdecl operator>> (ios _FAR & (_Cdecl *_f)(ios _FAR &) );
istream _FAR & _Cdecl operator>> (  signed char _FAR *);
:  :  :  :  :  :
istream _FAR & _Cdecl operator>> (unsigned char _FAR *);

ostream _FAR & _Cdecl operator<< (short);
ostream _FAR & _Cdecl operator<< (unsigned short);
ostream _FAR & _Cdecl operator<< (int);
:  :  :  :  :  :
ostream _FAR & _Cdecl operator<< (unsigned int);
```

După cum se vede, fişierul furnizează operatorii pentru tipuri de date obişnuite, ca *short, unsigned short* sau *int*. În Capitolul 1 s-au folosit extensiv stream-urile I/O *cin* şi *cout* pentru a realiza operaţii cu tastatura şi ecranul. În plus, în Capitolul 1 s-au discutat pe scurt şi stream-urile *cerr* şi *clog*, ce facilitau afişarea unor mesaje pe dispozitivul de eroare standard. Dacă examinaţi fişierul antet IOSTREAM.H, veţi găsi declaraţii similare celor ce urmează:

```
extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;
extern ostream_withassign _Cdecl cerr;
extern ostream_withassign _Cdecl clog;
```

Se poate vedea că stream-ul de intrare *cin* este definit folosind clasa *istream_withassign*. De asemenea, *cout, cerr* şi *clog* sunt definiţi folosind clasa stream de ieşire *ostream_withassign*. Cum se arată în Figura 11.2, cele două tipuri de clase sunt de fapt derivate din clasele *ostream* şi *istream*.

În general, obiectele clasei *withassign* acceptă operatorul de atribuire (=) care poate fi folosit pentru a atribui un stream altui stream sursă sau destinaţie. De

exemplu, stream-urile *cin* şi *cout* sunt asociate în mod normal cu tastatura şi ecranul. Dacă redirecţionaţi intrarea şi ieşirea programului, atunci conexiunea stream-urilor se va schimba.

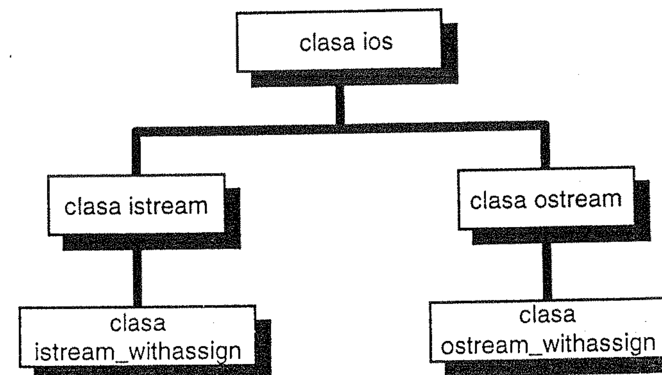


*Figura 11.2 Tipurile de clasă **ostream_withassign** şi **istream_withassign** sunt derivate din **ostream** şi **istream**.*

Următorul program, CHSTREAM.CPP, foloseşte operatorul de atribuire cu stream-ul *cout*, pentru a scrie ieşirea acestuia în fişierul COUT.DAT ce este în mod curent conectat la un stream fişier.

```
#include <fstream.h>

void main(void)
 {
   ofstream output("COUT.DAT");

   cout << "About to assign cout to the file" << endl;

   cout = output;

   cout << "This is not written to the screen, but rather, the file!"
        << endl;
 }
```

La compilarea şi execuţia programului, pe ecran va apărea primul mesaj. Al doilea mesaj nu va apărea pe ecran, ci va fi scris în fişierul COUT.DAT.

În mod analog, următorul program, COUT_PRN.CPP, atribuie stream-ul *cout* unui fişier ce corespunde dispozitivului de tipărire. La compilarea şi execuţia programului, acesta îşi va scrie ieşirea la dispozitivul de tipărire:

```
#include <fstream.h>

void main(void)
 {
   ofstream cprn("PRN");

   cout << "About to assign cout to the printer" << endl;

   cout = cprn;

   cout << "This is not written to the screen, but rather, the printer!"
        << endl;
 }
```



CHEIA SUCCESULUI

SĂ ÎNȚELEGEM STREAM-URILE I/O

Un *stream* este cel mai bine definit ca o serie de octeți ce se deplasează de la o sursă spre o destinație. Fișierul antet IOSTREAM.H definește în C++ clasele de stream I/O. Aceste clase sunt bazate pe clasa *ios* care furnizează variabilele și funcțiile membru necesare în majoritatea operațiilor de intrare/ieșire. Fișierul antet derivează apoi clasele *istream* și *ostream* care, la rândul lor, implementează operatorii de extracție (>>) și inserție (<<) folosiți pentru operațiile de intrare/ieșire. Majoritatea programelor C++ folosesc stream-urile I/O *cin, cout, cerr* și *clog*. Acest stream-uri sunt definite de clasele *istream_withassign* sau *ostream_withassign*, ceea ce indică faptul că obiectele de acest tip pot fi conectate cu alte obiecte de același tip folosind funcția membru operator de atribuire (=)

```
extern istream_withassign _Cdecl cin;

extern ostream_withassign _Cdecl cout;
```

CREAREA PROPRIILOR MANIPULATORI I/O

Așa cum ați învățat, un manipulator este o funcție ce poate fi plasată în interiorul unei operații de inserție sau extracție. În Capitolul 1 ați folosit manipulatori definiți în fișierul antet IOMANIP.H. Așa cum se va vedea, puteți crea propriii manipulatori pentru a-i folosi în stream-uri de intrare/ieșire. De exemplu, următoarea instrucțiune folosește un manipulator denumit *beep* ce activează difuzorul încorporat al calculatorului:

```
cout << beep << "Important message" << endl;
```

Următoarele instrucțiuni creează manipulatorul *beep*:


```
ostream& beep(ostream& stream)
{
  stream << '\007';
  return(stream);
}
```

După cum se vede, funcția returnează un obiect de tip *ostream*, de fapt același stream de ieșire pe care îl primește funcția drept parametru stream. Într-una din instrucțiunile funcției se inserează în stream caracterul ASCII de valoare 7. Funcția returnează apoi o referință la stream-ul actualizat. Când programul întâlnește un manipulator într-un operator de inserție sau de extracție, programul va apela funcția ce corespunde acestui manipulator. În instrucțiunea următoare, de exemplu, programul apelează funcția *beep* de două ori:

```
cout << beep << "Important message" << beep << endl;
```

Următorul program, BEEPMAN.CPP, folosește manipulatorul pentru activarea difuzorului înainte de afișarea unui mesaj:

```
#include <iostream.h>

ostream& beep(ostream& stream)
 {
   stream << '\007';
   return(stream);
 }

void main(void)
 {
   cout << beep << "This is an important message" << endl;
 }
```



CHEIA SUCCESULUI

CREAREA PROPRIILOR MANIPULATORI

În această carte s-au folosit manipulatori I/O prezentați în fișierul antet IOMANIP.H. În afara acestor manipulatori, programul își poate crea propriii manipulatori. Pentru a crea un manipulator, se declară o funcție care returnează o referință la un obiect de tip *istream* sau *ostream*. După cum se vede în continuare, primul parametru al funcției corespunde stream-ului:

Tip returnat

Nume manipulator

```
ostream&  manipulator_name(ostream& stream_name)
```

Parametru stream

În interiorul manipulatorului, funcţia realizează prelucrarea dorită apoi returnează stream-ul actualizat:

```
{
   // Desired statements

   // Return the updated stream
   return(stream_name);
}
```

Parametru stream returnat

Următorul program, TAB_MAN.CPP, foloseşte un manipulator denumit *tab*, care va insera un caracter de tabulare în stream-ul de ieşire:

```
#include <iostream.h>

ostream& tab(ostream& stream)
 {
   stream << '\t';
   return(stream);
 }

void main(void)
 {
   cout << "Hello" << tab << "world!" << endl;
 }
```

În Capitolul 1 aţi învăţat să folosiţi indicatorii *ios::scientific* şi *ios::fixed* pentru a selecta formatul de afişare a unei valori reale. De exemplu, următorul program, FIXEDSCI.CPP, foloseşte aceşti indicatori pentru a afişa valoarea 123.456:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   float value = 123.456;

   cout << setiosflags(ios::fixed) << value << endl;
   cout << setiosflags(ios::scientific) << value << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> FIXEDSCI  <ENTER>
123.456001
1.234560e+02
```

Dacă aceşti indicatori se folosesc frecvent, atunci este oportună crearea manipulatorilor *fixed* şi *scientific*, ca mai jos:


```
ostream& fixed(ostream& stream)
 {
   stream << setiosflags(ios::fixed);
   return(stream);
 }

ostream& scientific(ostream& stream)
 {
   stream << setiosflags(ios::scientific);
   return(stream);
 }
```

Următorul program, FIXSCI_2.CPP, foloseşte manipulatorii *fixed* şi *scientific* pentru afişarea valorii 123.456:

```
#include <iostream.h>
#include <iomanip.h>

ostream& fixed(ostream& stream)
 {
   stream << setiosflags(ios::fixed);
   return(stream);
 }

ostream& scientific(ostream& stream)
 {
   stream << setiosflags(ios::scientific);
   return(stream);
 }

void main(void)
 {
   float value = 123.456;

   cout << fixed << value << endl;
   cout << scientific << value << endl;
 }
```

Următorul program, DECOCTHE.CPP, creează şi foloseşte manipulatorii deja familiari, *dec*, *oct* şi *hex*:

```
#include <iostream.h>
#include <iomanip.h>

ios& dec(ios& stream)
 {
   stream.setf(ios::dec);
   return(stream);
 }
```

```
ios& oct(ios& stream)
 {
   stream.setf(ios::oct);
   return(stream);
 }

ios& hex(ios& stream)
 {
   stream.setf(ios::hex);
   return(stream);
 }

void main(void)
 {
   cout << oct << "Value 10 in octal is " << 10 << endl;
   cout << hex << "Value 10 in decimal is " << 10 << endl;
   cout << dec << "Value 10 in hexadecimal is " << 10 << endl;
 }
```

Următorul program, SHOWBASE.CPP, creează manipulatorul *showbase*, care va dirija stream-ul să insereze Ox în fața valorilor în hexazecimal și O în fața celor scrise în octal:

```
#include <iostream.h>
#include <iomanip.h>

ios& showbase(ios& stream)
 {
   stream.setf(ios::showbase);
   return(stream);
 }

void main(void)
 {
   cout << showbase;
   cout << oct << "Value 10 in octal is " << 10 << endl;
   cout << hex << "Value 10 in decimal is " << 10 << endl;
   cout << dec << "Value 10 in hexadecimal is " << 10 << endl;
 }
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SHOWBASE  <ENTER>
Value 10 in octal is 012
Value 10 in decimal is 0xa
Value 10 in hexadecimal is 10
```

În sfârșit, următorul program, SHOW_PO.CPP, creează manipulatorii *showpoint* și *showpos* care dirijează stream-ul pentru a afișa punctul la valorile reale, respectiv pentru a preceda numerele pozitive cu semnul plus:


```
#include <iostream.h>
#include <iomanip.h>

ios& showpoint(ios& stream)
 {
   stream.setf(ios::showpoint | ios::fixed);
   return(stream);
 }

ios& showpos(ios& stream)
 {
   stream.setf(ios::showpos);
   return(stream);
 }

void main(void)
 {
   cout << 10.0 / 2 << endl;
   cout << showpoint;
   cout << 10.0 / 2 << endl;

   cout << 1 << ' ' << 2 << ' ' << endl;
   cout << showpos;
   cout << 1 << ' ' << 2 << ' ' << endl;
 }
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SHOW_PO  <ENTER>
5
5.000000
1 2
+1 +2
```

MANIPULATORI CU PARAMETRI MULTIPLI

Fiecare din manipulatorii anteriori a fost destul de simplu, prin aceea că nu acceptă parametri. Următorul manipulator, denumit *spaces*, permite inserarea unui anumit număr de spații într-un stream de ieșire:

```
cout << spaces(5) << "Just skipped 5 spaces" << endl;
```

Pentru a crea un manipulator cu parametri, trebuie create două funcții. Prima funcție este foarte asemănătoare celor create până acum:

```
ostream& spaces(ostream& stream, int space_count)
 {
   for (int i = 0; i < space_count; i++)
```

```
    stream << ' ';

  return(stream);
}
```

Se observă că funcţia returnează o referinţă la un stream de ieşire şi primeşte tot o referinţă drept parametru. În plus, funcţia primeşte al doilea parametru de tip *int*, denumit *space_cout*. A doua funcţie foloseşte macroinstrucţiunea OMANIP, care este definită în fişierul antet IOMANIP.H, cum se arată mai jos:

```
OMANIP(int) spaces(int space_count)
{
  OMANIP(int) type(spaces, space_count);
  return(type);
}
```

În timpul compilării, preprocesorul va expanda macroinstrucţiunea, pentru a crea funcţia apelată de program la apariţia manipulatorului. După cum se observă, această funcţie apelează prima funcţie, specificând cei doi parametri.

Următorul program, SPACEMAN.CPP, foloseşte manipulatorul *spaces* pentru a insera un număr variabil de spaţii în stream-ul de ieşire:

```
#include <iostream.h>
#include <iomanip.h>

ostream& spaces(ostream& stream, int space_count)
{
  for (int i = 0; i < space_count; i++)
    stream << ' ';

  return(stream);
}

OMANIP(int) spaces(int space_count)
{
  OMANIP(int) type(spaces, space_count);
  return(type);
}

void main(void)
{
  for (int i = 1; i <= 5; i++)
    cout << "Skipping " << i << spaces(i) << "spaces" << endl;
}
```

***Observaţie:** Este posibil ca unele compilatoare să nu accepte total macroinstrucţiunea OMANIP. Dacă obţineţi erori de sintaxă la compilarea programului, apelaţi la un manual de referinţă al compilatorului.*

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> SPACEMAN <ENTER>
Skipping 1 spaces
Skipping 2  spaces
Skipping 3   spaces
Skipping 4    spaces
Skipping 5     spaces
```

În acelaşi mod, următorul program, NEWLINES.CPP, creează un manipulator denumit *newlines*, care dirijează stream-ul să tipărească un anumit număr de linii albe:

```
#include <iostream.h>
#include <iomanip.h>

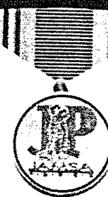
ostream& newlines(ostream& stream, int count)
{
  for (int i = 0; i < count; i++)
    stream << endl;

  return(stream);
}

OMANIP(int) newlines(int count)
{
  OMANIP(int) type(newlines, count);
  return(type);
}

void main(void)
{
  for (int i = 1; i <= 5; i++)
    cout << "Skipping " << i << newlines(i) << "lines" << endl;
}
```

***Observaţie:** Este posibil ca unele compilatoare să nu accepte total macroinstrucţiunea OMANIP. În caz de erori sintactice la compilarea programului, apelaţi la documentaţia compilatorului.*



MANIPULATORI CU PARAMETRI

La crearea propriilor manipulatori, este posibil de a transfera acestora o anumită valoare. Pentru a crea un manipulator cu parametru, trebuie, de fapt, create două funcţii. Prima funcţie specifică numele manipulatorului, parametrii şi instrucţiunile. De exemplu, următorul manipulator, *beeps*, specifică de câte ori se activează clopoţelul calculatorului:

```
ostream& bells(ostream& stream, int bell_count)
 {
   for (int i = 0; i < bell_count; i++)
     stream << '\a';

   return(stream);
 }
```

A doua funcţie foloseşte macrocomanda *OMANIP* definită în fişierul antet IOMANIP.H:

```
OMANIP(int) bells(int bell_count)
 {
   OMANIP(int) type(bells, bell_count);
   return(type);
 }
```

Pentru a activa clopoţelul de 3 ori, de exemplu, se va folosi manipulatorul astfel:

```
cout << bells(3);
```

În sfârşit, următorul program, SETFILL.CPP, creează un manipulator *setfill* de forma celor discutate în Capitolul 1:

```
#include <iostream.h>
#include <iomanip.h>

ostream& filler(ostream& stream, int fillchar)
 {
   stream.fill((char)fillchar);
   return(stream);
 }

OMANIP(int) filler(char fillchar)
 {
   OMANIP(int) type(filler, fillchar);
   return(type);
 }

void main(void)
 {
   cout << "Hello" << filler('.') << setw(15) << "world!" << endl;
 }
```

Observaţii: *Unele compilatoare nu acceptă macroinstrucţiunea OMANIP. În caz de erori sintactice, apelaţi la documentaţia compilatorului.*

La compilarea şi execuţia programului, pe ecran va apărea:

```
C:\> SETFILL  <ENTER>
Hello.........world!
```


Folosind această tehnică se poate crea o mare varietate de manipulatori. Încercaţi, de exemplu, să creaţi manipulatori care aliniază la stânga sau la dreapta ieşirea unui program.

CREAREA UNUI FIŞIER DE MANIPULATORI PROPRII

Aşa cum aţi învăţat, fişierul antet IOMANIP.H conţine prototipul funcţiilor câtorva manipulatori obişnuiţi. În funcţie de numărul şi complexitatea manipulatorilor folosiţi într-un program, s-ar putea dori plasarea acestor manipulatori (cu definiţiile complete) într-un fişier antet specific, cum ar fi MY_MANIP.H. Dacă nu doriţi ca şi alţi utilizatori să aibă acces la manipulatori, plasaţi prototipul acestora în fişierul antet şi codul efectiv al manipulatorilor într-o bibliotecă de obiecte. În Capitolul 17 găsiţi paşii care trebuie urmaţi pentru a crea o bibliotecă de clase. Aceiaşi paşi trebuie realizaţi şi în cazul creării unei biblioteci de manipulatori proprii.

SĂ ÎNŢELEGEM STREAM-URILE I/O ASOCIATE

În Capitolul 1 aţi învăţat că stream-urile I/O *cout* şi *clog* realizează o ieşire prin buffer. Prin urmare, ieşirea scrisă în aceste stream-uri nu apare pe ecran până când nu este îndeplinită una din condiţiile: bufferul este plin, programul se termină, programul goleşte bufferul, sau, în cazul lui *cout*, programul realizează o operaţie de intrare de la *cin*. În acest caz, se spune că *cin* este *asociat* cu stream-ul de ieşire *cout*. Dacă examinaţi fişierul antet IOSTREAM.H, veţi afla că clasa *ios* acceptă o funcţie membru denumită *tie*. Dacă programul apelează funcţia *tie* fără parametru, *tie* va returna un pointer la stream-ul de ieşire la care este asociat stream-ul de intrare. Următorul program, SHOWTIE.CPP, foloseşte funcţia *tie* pentru a verifica dacă stream-ul *cin* este asociat cu stream-ul *cout*:

```
#include <iostream.h>

void main(void)
 {
   if (*(cin.tie()) == cout)
     cout << "cin is tied to cout" << endl;
   else
     cout << "cin not tied to cout" << endl;
 }
```

Funcţia membru *tie* permite, de asemenea, asocierea dintre un stream de intrare şi unul de ieşire. Următorul program, TIE_CON.CPP, deschide un fişier ce corespunde dispozitivului consolă. Apoi, programul asociază stream-ul *cin* cu stream-ul fişier. După aceasta se scrie un mesaj în stream şi se aşteaptă 3 secunde. Când programul realizează operaţia de intrare pe *cin*, mesajul va fi afişat fără întârziere pe *cout*:

```
#include <fstream.h>
#include <time.h>

void main(void)
 {
   time_t start_time, current_time;

   ofstream screen("CON");

   cin.tie(&screen);

   screen << "Hello C++ world!--Press Enter to continue";
   time(&start_time);

   do {
     time(&current_time);
   } while ((current_time - start_time) < 3);

   cin.get();
   screen.close();
 }
```

Se poate, de asemenea, asocia un stream de ieşire cu un alt stream de ieşire. În acest mod, când bufferul de ieşire este golit, la fel va fi şi bufferul de ieşire al stream-ului asociat. În mod prestabilit, C++ asociază *cerr* la *cout* şi *clog* la *cout*. Dacă *cerr* sau *clog* este golit, la fel va fi şi *cout*. Următorul program, SHOWTIES.CPP, arată cum sunt asociate stream-urile în mod prestabilit în C++:

```
#include <iostream.h>

void main(void)
 {
   if (*(cin.tie()) == cout)
     cout << "cin is tied to cout" << endl;

   if (*(cerr.tie()) == cout)
     cout << "cerr is tied to cout" << endl;

   if (*(clog.tie()) == cout)
     cout << "clog is tied to cout" << endl;

   if (*(cout.tie()) == NULL)
     cout << "cout is not tied" << endl;
 }
```

La compilarea şi execuţia acestui program, ecranul va afişa:

```
C:\> SHOWTIES  <ENTER>
cin is tied to cout
cerr is tied to cout
```



```
clog is tied to cout
cout is not tied
```

MAI MULTE DESPRE INTERACŢIUNEA CLASELOR DE STREAM-URI

În Capitolul 3 aţi examinat stream-urile fişier în C++. Aşa cum aţi învăţat, fişierul antet FSTREAM.H defineşte clasele pentru stream-uri fişier *ifstream*, *ofstream* şi *fstream*. Când în programe trebuie să se realizeze operaţii de intrare pe fişiere, atunci se creează obiecte de tipul *ifstream*. Când se execută operaţii de ieşire, se folosesc obiecte de tipul *ofstream*. În sfârşit, când acelaşi fişier trebuie citit şi scris, programele folosesc obiecte de tipul *fstream*. Dacă examinaţi declaraţiile acestor clase din fişierul FSTREAM.H, veţi vedea că aceste clase se bazează pe clasele *istream*, *ostream* şi *iostream*, cum se arată în Figura 11.3.

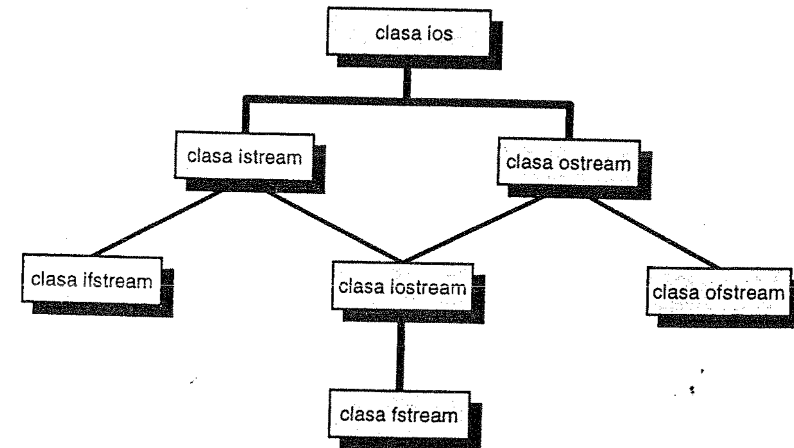


***Figura 11.3.** Clasele **ifstream, ofstream** şi **fstream** sunt derivate din clasele **istream, ostream** şi **iostream**.*

Clasele *ifstream* şi *ofstream* adaugă la clasele de bază din care provin funcţiile membru *open* şi *close*. Clasa *iostream* adaugă facilitatea de citire şi scriere la clasele moştenite *istream* şi *ostream*. Clasa *fstream* adaugă funcţiile *open* şi *close* la cele furnizate deja de clasa *iostream*.

SĂ ÎNŢELEGEM IEŞIREA FORMATATĂ ŞI NEFORMATATĂ

Când realizaţi operaţii I/O, se pot folosi clase pentru intrări/ieşiri formatate sau neformatate. Toate clasele discutate în acest capitol au realizat I/O formatat. Tabela 11.1 prezintă clasele pentru I/O formatate.

Numele clasei	Descriere
ios	Clasa de bază pentru stream-uri de intrare şi ieşire
istream	Derivată din *ios*, adaugă operatorul de extracţie
ostream	Derivată din *ios*, adaugă operatorul de inserţie
iostream	Derivată din *istream* şi *ostream*, adaugă operaţiile de citire şi scriere
istream_withassign	Derivată din *istream*, adaugă operatorul de atribuire
ostream_withassign	Derivată din *ostream*, adaugă operatorul de atribuire
ifstream	Derivată din *istream*, adaugă metodele de închidere şi deschidere
ofstream	Derivată din *ostream*, adaugă metodele de închidere şi deschidere
fstream	Derivată din *iostream*, adaugă metodele de închidere şi deschidere
stdiostream	Derivată din *ios*, adaugă o conexiune la un fişier din *stdio*

Tabela 11.1. *Clasele C++ pentru I/O formatat*

Când programele realizează intrări/ieşiri formatate, clasele I/O convertesc valorile întregi şi reale în caractere ASCII corespunzătoare. De exemplu, următoarea instrucţiune tipăreşte valorile 22, 33 şi 44:

```
cout << 22 << " " << 33 << " " << 44 << endl;
```

Pentru a afişa valoarea 22, de exemplu, programul tipăreşte valoarea ASCII 50 (valoarea pentru caracterul '2') de două ori. Analog, pentru a tipări valoarea 33, programul va tipări valoarea ASCII 51 de două ori.

Când programele realizează afişare *neformatată*, programul tipăreşte reprezentarea binară a fiecărei valori, fără conversii. Tabela 11.2 prezintă clasele C++ pentru I/O neformatat.

Numele clasei	Descriere
streambuf	Clasa de bază pentru I/O prin buffer care implementează pointerii de citire şi scriere şi operaţii I/O simple
filebuf	Derivată din *streambuf*, adaugă metodele de deschidere şi închidere
stdiobuf	Derivată din *streambuf*, adaugă o conexiune la un fişier din *stdio*

Tabela 11.2 *Clasele C++ pentru I/O neformatat*

Pentru a înţelege mai bine ierarhia de clase pentru I/O neformatat, să privim diagrama din Figura 11.4.

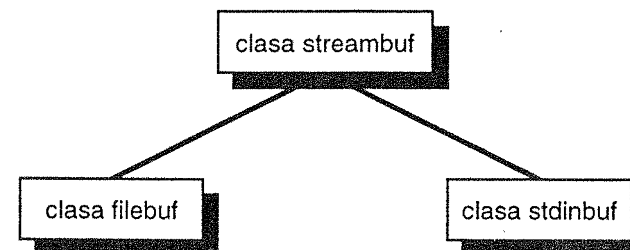


Figura 11.4. *Clasele C++ pentru I/O neformatat.*

Fişierul antet IOSTREAM.H defineşte clasele pentru I/O neformatat. Când programele realizează I/O neformatat, se lucrează în principiu cu date în buffer, cum se arată în continuare.

SĂ ÎNŢELEGEM INTRĂRILE/IEŞIRILE PRIN BUFFER

Clasa *streambuf* oferă baza pentru operaţii I/O în C++ folosind buffer-ul. În asemenea situaţii, programele citesc sau scriu informaţii într-un anumit buffer (o zonă tampon de memorie), cum se arată în Figura 11.5.

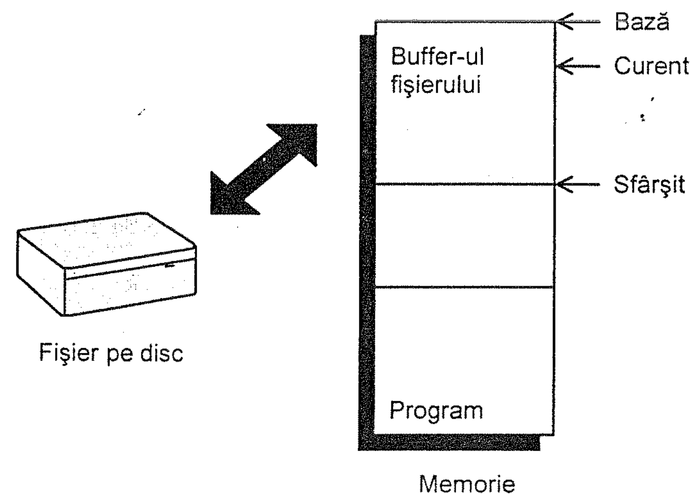


Figura 11.5. *I/O prin buffer plasează datele în buffer-ul de memorie*

După cum se observă, pentru controlul buffer-ului se folosesc câţiva pointeri. Dacă examinaţi fişierul antet IOSTREAM.H, veţi vedea că în fişier sunt definite diferite funcţii membru ale claselor. Multe din aceste funcţii manipulează pointeri la buffere sau caractere referite de pointeri. Nu se pot crea obiecte de tipul *streambuf*. În schimb, se pot deschide fişiere folosind clasa *filebuf*. În majoritatea claselor, însă, programele nu vor realiza I/O prin buffer folosind obiecte de tipul *filebuf*, ci vor folosi obiecte de tipul *ifstream* sau *ofstream*, aşa cum am văzut în Capitolul 3. Clasa *ios* foloseşte un pointer la un obiect de tipul *streambuf* pentru a putea realiza I/O prin buffer. Este important de reţinut că clasa *ios* nu derivă din clasa *streambuf*, dar foloseşte, în schimb, un pointer la un obiect de tipul *streambuf*.

Pentru a manipula obiecte ale clasei *filebuf*, trebuie să înţelegem mai întâi scopul funcţiilor membru ale clasei *streambuf*. Dacă examinaţi fişierul IOSTREAM.H, veţi găsi prototipurile unor funcţii membru de forma următoare:

```
int  _Cdecl sgetc();                    // Peek at next character in
                                        // the buffer
int  _Cdecl snextc();                   // Advance to and return next
                                        // character in the buffer
int  _Cdecl sbumpc();                   // Return the current
                                        // character in the buffer
                                        // and advance the pointer
void _Cdecl stossc();                   // Advance to the next
                                        // character in the buffer
int  _Cdecl sgetn(char _FAR *, int);    // Get next n characters in
                                        // the buffer
int  _Cdecl sputbackc(char);            // Return a character to the
                                        // buffer
int  _Cdecl in_avail();                 // Number of available
                                        // characters in the buffer
int  _Cdecl sputc(int);                 // Put one character into the
                                        // buffer
int  _Cdecl sputn(const char _FAR *, int); // Put n characters into
                                           // the buffer
int  _Cdecl out_waiting();              // Number of unflushed
                                        // characters in output
                                        // buffer
```

Pe lângă funcţiile membru prezentate anterior, clasa *streambuf* defineşte funcţii care permit căutarea sau deplasarea în buffer şi funcţii care sincronizează bufferul pentru operaţii de intrare/ieşire. La realizarea operaţiilor prin buffer cu fişiere folosind obiecte de tipul *filebuf*, se folosesc una sau mai multe din aceste funcţii. Pentru a le înţelege mai bine, Tabela 11.2 listează scopul funcţiilor şi valoarea returnată de acestea.

Nume funcţie	Scop
sgetc	Anticipează următorul caracter din buffer. Dacă aceasta nu există, se returnează EOF. Pointerul *get* nu este deplasat.
snextc	Deplasează pointerul *get* înainte cu un caracter şi returnează următorul caracter.
sbumpc	Returnează caracterul curent apoi avansează pointerul *get*. Dacă caracterul nu există, se returnează EOF.
stossc	Deplasează pointerul *get* înainte, cu un caracter.
sgetn	Returnează următoarele *n* caractere din buffer. Funcţia întoarce numărul de caractere returnate. Dacă nu există caractere, se returnează EOF.
sputbackc	Plasează înapoi în buffer ultimul caracter luat din buffer.
in_avail	Returnează numărul de caractere disponibile.
sputc	Plasează caracterul specificat în buffer şi avansează pointerul *put*.
sputn	Plasează *n* caractere dintr-un şir în buffer, avansând pointerul *put*.
out_waiting	Returnează numărul de caractere din bufferul de ieşire.

*Tabela 11.3 Scopul funcţiilor membru **filebuf**.*

DESCHIDEREA UNUI FIŞIER PENTRU OPERAŢII I/O PRIN BUFFER

În majoritatea cazurilor, programele folosesc clasele *ifstream* şi *ofstream* pentru operaţii I/O cu fişiere. Însă, prin examinarea operaţiilor cu fişiere ce folosesc clasa *filebuf* veţi înţelege mai bine operaţiile I/O prin buffer. Deschiderea unui fişier pentru operaţii I/O prin buffer folosind clasa *filebuf* este foarte asemănătoare cu deschiderea unui stream fişier pentru operaţii de intrare/ieşire. Mai întâi, se declară un obiect de tipul *filebuf*, ca mai jos:

```
filebuf buffer_file;
```

Apoi, se deschide fişierul folosind funcţia membru *open*, astfel:

```
buffer_file.open("FILENAME.EXT", ios::in);
```

Următorul program, OPENBUF.CPP, deschide un fişier specificat în linia de comandă pentru intrare prin buffer. Dacă fişierul va fi deschis cu succes, programul afişează un mesaj, iar apoi fişierul se închide. Dacă fişierul nu este deschis, programul afişează un mesaj de eroare corespunzător:

```
#include <fstream.h>

void main(int argc, char **argv)
```

```
{
  filebuf buffered_file;

  buffered_file.open(argv[1], ios::in);

  if (buffered_file.is_open())
   {
     cout << "File successfully opened" << endl;
     buffered_file.close();
   }
  else
    cerr << "Error opening the file: " << argv[1] << endl;
}
```

După cum se poate vedea, programul foloseşte metoda *close* pentru a închide fişierul când nu va mai fi necesar. Următorul program, SHOWBUF.CPP, deschide un fişier pentru intrare prin buffer. Programul afişează apoi conţinutul fişierului, caracter cu caracter:

```
#include <fstream.h>

void main(int argc, char **argv)
 {
   filebuf buffered_file;

   int letter;

   buffered_file.open(argv[1], ios::in);

   if (buffered_file.is_open())
    {
      while ((letter = buffered_file.sgetc()) != EOF)
       {
         cout.put((char)letter);
         buffered_file.snextc();
       }

      buffered_file.close();
    }
   else
     cerr << "Error opening the file: " << argv[1] << endl;
 }
```

Programul foloseşte funcţia membru *sgetc* pentru a anticipa următorul caracter din buffer. Dacă sfârşitul de fişier nu a fost întâlnit, programul afişează caracterul şi apoi avansează pointerul la următorul caracter, folosind funcţia *snextc*.

În mod similar, următorul program, EASYSHOW.CPP, deschide un fişier aflat în buffer. Programul afişează conţinutul fişierului, folosind operatorul de inserţie pentru a afişa bufferul fişierului. Dacă examinaţi fişierul antet IOSTREAM.H, veţi

descoperi că operatorul de inserţie este suprapus pentru operaţiile care folosesc o adresă la *streambuf*. Funcţia corespunzătoare acestui operator extrage caracterele din buffer, plasându-le în stream-ul de ieşire:

```
#include <fstream.h>

void main(int argc, char **argv)
 {
   filebuf buffered_file;
   buffered_file.open(argv[1], ios::in);
   if (buffered_file.is_open())
    {
      cout << &buffered_file;
      buffered_file.close();
    }
   else
     cerr << "Error opening the file: " << argv[1] << endl;
 }
```

IMAGINE DE ANSAMBLU

Pentru a înţelege cum lucrează diferitele clase I/O, trebuie avut în vedere că clasele I/O conţin funcţii publice ce pot fi apelate de programe şi membri de tip *private* ce controlează operaţiile I/O. La deschiderea unui fişier pentru intrare, de exemplu, este creat un obiect de tipul *ifstream*. Aşa cum aţi citit, clasa *ifstream* (care adaugă membrii *open* şi *close*) este bazată pe clasa *istream*. Clasa *istream*, la rândul ei, este derivată din clasa de bază *ios*, adăugând la aceasta operatorul de extracţie. Când programele realizează operaţii de intrare folosind stream-ul, datele trebuie plasate în anumite zone ale memoriei. Cum s-a arătat, datele sunt plasate într-un buffer de memorie ce corespunde unui obiect *streambuf*. Obiectul *streambuf* conţine pointeri şi funcţii ce controlează accesul la date. Ţineţi minte, totuşi, că clasa *ios* nu este derivată din clasa *streambuf*, ci foloseşte un pointer la un obiect *streambuf*.

REZUMAT

Aproape toate programele C++ folosesc intens stream-uri I/O şi stream-uri fişiere. Acest capitol a analizat în amănunt stream-urile I/O - atât modul lor de folosire, cât şi relaţiile dintre diferite tipuri de clase. Înainte de a trece la Capitolul 12, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Fişierul antet IOSTREAM.H defineşte clasele de stream I/O. În cadrul fişierului, găsiţi o serie de definiţii de clase asociate.
- ✓ Clasa *ios* defineşte clasa de bază pentru operaţii I/O. Clasele *istream* şi *ostream* adaugă la clasa *ios* operatorii de inserţie şi

extracţie. În cadrul fişierului IOSTREAM.H, găsiţi funcţiile membru ce definesc aceste operaţii.

✓ Stream-ul *cin* este definit ca *istream_with assign*. Analog, stream-urile *cout, cerr* şi *clog* sunt definite ca *ostream_with assign*. Clasele definite ca *_withassign* conţin operatorul de atribuire (=), ce poate fi folosit pentru a atribui un alt stream unui obiect al clasei.

✓ C++ implementează manipulatorii prin funcţii, în acelaşi mod în care programele realizează suprapunerea operatorilor. Pentru a crea un manipulator pentru un stream I/O, se creează o funcţie ce primeşte o referinţă la un stream drept parametru, apoi se manipulează stream-ul într-un anumit mod. Funcţia returnează apoi referinţa la stream.

✓ Se pot crea şi manipulatori cu parametri. Cel mai simplu mod de a crea un astfel de manipulator este de a folosi macroinstrucţiunea *OMANIP* definită în fişierul antet IOMANIP.H.

✓ Dacă se creează mai mulţi manipulatori, aceştia se pot salva într-un fişier antet, simplificând folosirea lor în alte programe.

✓ Pentru un bun control al buffer-ilor I/O, programele pot *asocia* un stream cu alt stream. De exemplu, stream-ul *cin* este asociat cu *cout*. Dacă stream-ul *cout* conţine ieşirea prin buffer care nu a fost încă afişată, atunci aceasta va apărea imediat ce se realizează o intrare de la *cin*.

✓ Fişierul antet FSTREAM.H defineşte stream-urile fişier în C++. În cadrul acestui fişier, găsiţi definiţiile claselor *ifstream, ofstream* şi *fstream*. Aceste clase adaugă metodele *open* şi *close* la cele moştenite din clasele *istream, ostream* şi *iostream*.

✓ Când programele realizează operaţii I/O formatate, informaţia de ieşire este mai întâi convertită în formatul ASCII. De asemenea, informaţia de intrare este convertită din ASCII în formatul dorit. Operaţiile I/O neformatate, pe de altă parte, nu convertesc valorile.

✓ Pentru a realiza operaţii I/O neformatate, programele pot folosi obiecte *Filebuf*.

CAPITOLUL 12
APROFUNDAREA STREAM-URILOR ŞIR

În cuprinsul acestei cărţi programele folosesc extensiv stream-urile I/O pentru a realiza operaţii cu tastatura, ecranul sau fişierele. C++ furnizează, de asemenea, un set de *stream-uri şir* ce pot fi folosite pentru păstrarea caracterelor ce urmează a fi afişate formatat sau pentru conversii numerice. De exemplu, dacă un şir conţine reprezentarea ASCII a numărului 1001, se poate folosi un stream pentru a converti caracterele într-o valoare de tip *int*. Acest capitol examinează stream-urile şir în detaliu. Aşa cum veţi vedea, folosirea acestor stream-uri este asemănătoare multor operaţii efectuate cu stream-urile I/O. La terminarea acestui capitol, veţi cunoaşte:

- Ce este un stream şir şi ce realizează
- Unde se găsesc definiţiile claselor pentru stream-uri şir
- Care sunt cele mai importante clase pentru stream-uri şir
- Ce sunt stream-urile şir de intrare
- Ce sunt stream-urile şir de ieşire
- Cum acceptă stream-urile funcţii membru
- Ce este un buffer dinamic

SĂ ANALIZĂM FIŞIERUL STRSTREA.H

C++ defineşte clasele de stream-uri şir în fişierul antet STRSTREA.H. Tipăriţi acum o copie a conţinutului fişierului. Puteţi folosi conţinutul fişierului ca o referinţă în cursul discuţiilor din acest capitol.

Pentru a folosi un stream şir, trebuie să includeţi fişierul antet la începutul programului, cum se arată mai jos:

```
#include <strstrea.h>
```

Dacă examinaţi conţinutul fişierului, veţi vedea că se definesc o serie de clase. Cele mai cunoscute 3 clase de stream şir sunt descrise pe scurt în Tabela 12.1. În secţiunile următoare aceste clase vor fi descrise în detaliu:

Stream-ul	Scopul
istrstream	Suportă operaţii de formatare a şirurilor de intrare pentru şiruri membru
ostrstream	Suportă operaţii de formatare a şirurilor de ieşire pentru şiruri din memorie
strstream	Suportă operaţii de formatare a şirurilor de intrare sau ieşire pentru şiruri din memorie

Tabela 12.1. *Clasele de stream-uri şir definite în fişierul antet STRSTREA.H*

FOLOSIREA STREAM-URILOR ŞIR DE INTRARE

Cel mai bun mijloc de a înţelege stream-urile şir este de a examina câteva programe simple care folosesc aceste stream-uri. Stream-urile şir acceptă operaţii de intrare sau ieşire. În general, un stream şir de intrare acceptă operatorul de extracţie (>>), permiţând programelor să introducă variabile din buffer-ul stream-ului. Un stream şir de ieşire, pe de altă parte, acceptă operatorul de inserţie (<<). Prin folosirea acestuia, programele pot scrie în stream caractere, şiruri, numere, şi chiar caractere speciale. Stream-urile şir de intrare sunt adesea folosite pentru a converti o reprezentare ASCII a unui număr, la o reprezentare de tip întreg sau real. Stream-urile şir de intrare sunt adesea folosite pentru a converti o reprezentare a unui număr, la o valoare *int* sau *float*. De exemplu, să presupunem că şirul de caractere *number* conţine valoarea 1.2345, cum se arată în Figura 12.1.

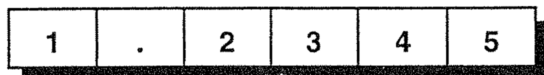


Figura 12.1. *Reprezentarea ASCII a unei valori reale*

Înainte de a lucra cu valoarea (de a executa operaţii matematice asupra acesteia), programul trebuie mai întâi să converteascǎ valoarea ASCII în real. În limbajul C, s-ar putea folosi funcţia de bibliotecă *atof*, aşa cum se arată în următorul program, ATOF.C:

```
#include <stdlib.h>
#include <stdio.h>

void main(void)
 {
   char *number = "1.2345";
   float value;
```



```
   value = atof(number);

   printf("The value is %f\n", value);
   printf("The value squared is %f\n", value * value);
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> ATOF <ENTER>
The value is 1.234500
The value squared is 1.523990
```

În cadrul programelor C++ se pot încă folosi funcţii ca *atof* pentru a realiza asemenea conversii.

Partea neplăcută a folosirii acestor funcţii este că trebuie reţinut numele funcţiilor (ca *atoi, atof, atol* etc.) De aceea, unii programatori C folosesc funcţia *sscanf* pentru a citi valori dintr-un şir de caractere, eliminând necesitatea de a memora nume de funcţii. Următorul program, SSCANF.C, citeşte o valoare reală dintr-un şir ASCII, realizând conversia dorită:

```
#include <stdio.h>

void main(void)
 {
   char *number = "1.2345";
   float value;

   sscanf(number, "%f", &value);

   printf("The value is %f\n", value);
   printf("The value squared is %f\n", value * value);
 }
```

Conversia valorilor folosind *sscanf* este foarte asemănătoare cu citirea valorilor dintr-un stream şir de intrare. Diferenţa, totuşi, este aceea că la stream-urile şir nu trebuie folosit specificatorul de format, cum este % f. În locul acestuia se foloseşte operatorul de extracţie (>>). Următorul program, INPUTSTR.CPP, ilustrează modul în care se foloseşte un stream şir pentru a converti o valoare reală:

```
#include <strstrea.h>
#include <iostream.h>

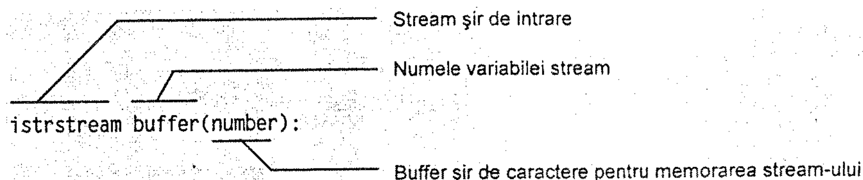
void main(void)
 {
   char *number = "1.2345";
   float value;

   istrstream buffer(number);
```

```
    buffer >> value;

    cout << "The value is " << value << endl;
    cout << "The value squared is " << value * value << endl;
 }
```

Programul creează un stream şir de intrare, denumit *buffer*, folosind următoarea instrucţiune:



La transmiterea unui singur buffer şir de caractere constructorului clasei *istrstream*, se presupune că lungimea buffer-ului şir (primul parametru) corespunde caracterului NULL de terminare a şirului. Următoarele instrucţiuni, pe de altă parte, creează un buffer de 256 octeţi:

```
char °ir[256];

istrstream buffer(°ir, sizeof(°ir));
```

Dacă se specifică lungimea buffer-ului în timpul creării obiectului, stream-ul va accepta numărul de caractere specificat.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM STREAM-URILE ŞIR DE INTRARE

Un stream şir este un buffer de memorie ce conţine caractere. Stream-urile şir de intrare sunt denumite astfel deoarece programele pot folosi operatorul de extracţie (>>) pentru a atribui valori din aceste stream-uri unor variabile. Pentru a crea un stream de intrare, trebuie declarat un obiect de tipul *istrstream*, ca mai jos:

```
char °ir[256];
char *book = "Success with C++";

istrstream buffer(°ir, sizeof(°ir));
istrstream  text(book);
```

Prima declaraţie de stream (*buffer*) specifică dimensiunea buffer-ului corespunde stream-ului. În cazul celei de a doua declaraţie (*text*), dimensiunea buffer-ului corespune terminatorului NULL de la sfârşitul şirului „Succes cu C++". Folosind operatorul de extracţie, programele pot atribui date din aceste stream-uri unor variabile, cum se arată mai jos:


```
buffer >> value;
text >> word;
```

Stream-urile şir de intrare sunt frecvent folosite pentru a converti numere în reprezentarea ASCII în echivalentele lor întregi sau reale.

CITIREA MAI MULTOR VALORI DINTR-UN STREAM ŞIR DE INTRARE

La citirea datelor din stream-uri de intrare, cum sunt *cin* sau stream-uri fişier, se pot citi mai multe valori la un moment dat folosind operatorul de extracţie:

```
cout << "Type your first and last name: ";

cin >> first >> last;
```

Când se lucrează cu stream-uri şir de intrare, este posibil ca buffer-ul şirului să conţină mai multe valori. De exemplu, buffer-ul şir ar putea conţine valorile 1, 100 şi 1001, cum se arată în Figura 12.2

1		1	0	0		1	0	0	1

***Figura 12.2** Valori întregi multiple într-un buffer de stream şir*

Folosind operatorul de extracţie, se pot citi toate cele 3 valori astfel:

```
int a, b, c;

buffer >> a >> b >> c;
```

Următorul program, THREEVAL.CPP, foloseşte operatorul de extracţie pentru a citi 3 valori dintr-un buffer şir:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *°ir = "1 100 1001";

   istrstream buffer(°ir);

   int a, b, c;

   buffer >> a >> b >> c;
```

```
    cout << "The values are " << a << ' ' << b << ' ' << c << endl;
}
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> THREEVAL <ENTER>
The values are 1 100 1001
```

În cazul precedent, programul a citit 3 variabile folosind o singură instrucţiune, dar şi varianta următoare este corectă:

```
buffer >> a;
buffer >> b;
buffer >> c;
```

În anumite situaţii, nu se cunoaşte numărul de valori conţinute în buffer. Dacă programul încearcă să citească un buffer gol, stream-ul va activa indicatorul *fail*. Următorul program, STR_FAIL.CPP, citeşte valori întregi dintr-un stream şir de intrare, până când stream-ul devine vid:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *°ir = "1 2 3 4 5 6 7 8 9";

   int value;

   istrstream buffer(°ir);

   while (! buffer.fail())
     {
       buffer >> value;
       if (! buffer.fail())
         cout << value << endl;
     }
 }
```

După cum se poate vedea, programul ciclează până când se activează indicatorul *fail* al stream-ului. În cadrul ciclului, să observăm că programul testează starea indicatorului după fiecare operaţie de intrare. Dacă intrarea reuşeşte, programul tipăreşte valoarea citită. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> STR_FAIL <ENTER>
1
2
3
4
5
```



```
6
7
8
9
```

În programul precedent, buffer-ul conţinea mai multe valori de tip *int*. În unele cazuri însă, este posibil ca buffer-ul să conţină valori de tipuri diferite. De exemplu, următorul program, DIFFTYPE.CPP, citeşte valori de tip *int*, *float* şi *long* dintr-un stream şir de intrare:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *numbers = "1001 1.2345 123456789L";

   int count;
   float rate;
   long distance;

   istrstream buffer(numbers);

   buffer >> count;
   buffer >> rate;
   buffer >> distance;

   cout << "The int count is " << count << endl;
   cout << "The float rate is " << rate << endl;
   cout << "The long distance is " << distance << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran, va apărea:

```
C:\> DIFFTYPE <ENTER>
The int count is 1001
The float rate is 1.2345
The long distance is 123456789
```

Când programele folosesc operatorul de extracţie pentru a citi date dintr-un stream şir de intrare, compilatorul determină tipul valorii de extras după tipul variabilei. De exemplu, în programul precedent, compilatorul ştia să extragă mai întâi o valoare de tipul *int*, urmată de *float* şi *long*. Pentru a realiza aceste operaţii, compilatorul apelează o funcţie care citeşte şi converteşte valoarea în tipul de date corect. În anumite situaţii, însă, valorile din buffer pot să nu corespundă celor pe care le necesită programul. De exemplu, să presupunem că, în loc să conţină valorile:

"1001 1.2345 123456789L"

buffer-ul începe cu un şir de caractere:

"ABC 1001 1.2345 123456789L"

Când funcţia apelată pentru conversia unei valori de un anumit tip întâlneşte un caracter incorect, ea va opri imediat conversia, returnând valoarea creată până la punctul de eroare. Funcţia nu va avansa pointerul dincolo de punctul de eroare. Dacă primul caracter din stream este invalid, funcţia va atribui indicatorul *fail* al stream-ului. Următorul program, BAD_DATA.CPP, ilustrează modul de folosire a funcţiei *fail* pentru a determina dacă o operaţie I/O a reuşit:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *numbers = "ABC 1001 1.2345 123456789L";

   int count;
   float rate;
   long distance;

   istrstream buffer(numbers);

   buffer >> count;

   if (buffer.fail())
    {
     cerr << "Error reading the int value for count" << endl;
     buffer.clear();
    }

   buffer >> rate;

   if (buffer.fail())
    {
     cerr << "Error reading the float value for rate" << endl;
     buffer.clear();
    }

   buffer >> distance;

   if (buffer.fail())
    {
     cerr << "Error reading the long value for distance" << endl;
     buffer.clear();
    }
   cout << "The int count is " << count << endl;
   cout << "The float rate is " << rate << endl;
   cout << "The long distance is " << distance << endl;
 }
```


După cum se observă, programul foloseşte funcţia membru *fail* pentru a testa reuşita fiecărei operaţii I/O. Dacă are loc o eroare, programul afişează mesajul de eroare şi apoi dezactivează indicatorul. Dacă o funcţie de conversie întâlneşte un caracter incorent, funcţia opreşte conversia fără a avansa pointerul la stream. Astfel, toate cele 3 operaţii de intrare au eşuat la întâlnirea literei A. Dacă compilaţi şi executaţi acest program, pe ecran va apărea:

```
C:\> BAD_DATA <ENTER>
Error reading the int value for count
Error reading the float value for rate
Error reading the long value for distance
The int count is 0
The float rate is 1.3705e-034
The long distance is 0
```

Se observă că programul afişează un mesaj de eroare după fiecare operaţie I/O, iar valorile atribuite variabilelor nu sunt corecte.



CITIREA MAI MULTOR VALORI DINTR-UN STREAM ŞIR DE INTRARE

Un stream şir de intrare este un buffer de caractere din care programele pot citi valori folosind operatorul de extracţie. În funcţie de conţinutul stream-ului, programele pot citi valori multiple dintr-un stream într-o singură instrucţiune:

```
buffer >> day >> month >> year;
```

Când compilatorul C++ întâlneşte o operaţie I/O pe stream, el apelează o funcţie pentru a extrage o valoare de un anumit tip. Dacă funcţia se execută cu succes, ea converteşte valoarea reprezentată în ASCII într-o valoare având tipul variabilei asociate. Dacă în timpul conversiei apare o eroare, funcţia activează indicatorul *fail* al stream-ului şi opreşte imediat conversia, returnând valoarea creată până în acel moment. Funcţia nu va avansa pointerul la stream dincolo de caracterul incorect.

FOLOSIREA FUNCŢIILOR MEMBRU ALE STREAM-ULUI ŞIR DE INTRARE

Programele anterioare au folosit operatorul de extracţie pentru a citi valori dintr-un stream şir de intrare. Utilizarea cea mai frecventă a unui stream şir de intrare este de a converti o valoare din reprezentarea ei ASCII. Totuşi, în funcţie de necesităţile programului şi conţinutul stream-ului, pot exista situaţii când se doreşte citirea unui singur caracter din stream, sau a unui buffer întreg. Următorul program, IN_GET.CPP, citeşte conţinutul unui stream caracter cu caracter, folosind funcţia membru *get*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *°ir = "Success with C++";

   istrstream buffer(°ir);

   for (char letter = buffer.get(); ! buffer.eof();
        letter = buffer.get())
     cout.put(letter);
 }
```

Aşa cum se vede, programul foloseşte funcţia membru *get* pentru a citi un caracter din stream şi funcţia *eof* pentru a determina dacă stream-ul este vid.

În funcţie de informaţia conţinută în buffer-ul stream-ului, apare câteodată necesitatea de a citi informaţiile situate după o serie de caractere, înainte de a începe o operaţie de extracţie. Folosind funcţia *get* în acest mod, se poate citi conţinutul stream-ului caracter cu caracter. Într-un mod similar, următorul program, READALL.CPP, citeşte în totalitate buffer-ul unui stream folosind funcţia membru *getline*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char *°ir = "Success with C++";

   istrstream buffer(°ir);

   char book[256];

   buffer.getline(book, sizeof(book));

   cout << book << endl;
 }
```

FOLOSIREA STREAM-URILOR ŞIR DE IEŞIRE

Aşa cum convertim reprezentări ASCII în valori de tip *int* sau *float* folosind stream-uri şir de intrare, la fel putem converti valorile dintr-un şir. În limbajul C, programatorii folosesc funcţii ca *itoa* (*int* în ASCII) pentru a converti o valoare numerică în ASCII. Următorul program, ITOA.C, ilustrează o astfel de conversie:

```
#include <stdio.h>
#include <stdlib.h>
```



```
void main(void)
 {
   char buffer[256];

   int value = 1001;

   itoa(value, buffer, 10);

   printf("Value in ASCII is %s\n", buffer);
 }
```

În alte programe C se foloseşte funcţia *sprintf* pentru a crea o reprezentare ASCII a unei valori. Următorul program, SPRINTF.C, foloseşte funcţia *sprintf* pentru a crea reprezentarea ASCII a valorii 1.2345:

```
#include <stdio.h>

void main(void)
 {
   char buffer[256];

   float value = 1.2345;

   sprintf(buffer, "%f", value);

   printf("The value in ASCII is %s\n", buffer);
 }
```

În limbajul C++, valorile pot fi convertite în ASCII folosind stream-urile şir de ieşire. Crearea unui stream şir de ieşire se face la fel cu cea a unui stream şir de intrare, declarând un obiect de tipul *ostrstream* la care se asociază un buffer, cum se arată mai jos:

```
char °ir[256];

ostrstream buffer(°ir, sizeof(°ir));
```

Denumirea de *stream-uri şir de ieşire* derivă de la faptul că programele pot folosi operatorul de inserţie pentru a plasa caractere ASCII în aceste stream-uri. Următorul program, OUT_STR.CPP ilustrează modul în care s-ar putea folosi stream-urile şir de ieşire:

```
#include <strstrea.h>
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   char °ir[256];
   ostrstream buffer(°ir, sizeof(°ir));
```

```
  float price = 29.95;
  int programs = 300;

  buffer << "Success with C++ $";
  buffer << setprecision(2) << price << " with over ";
  buffer << programs << " programs" << ends;

  cout << şir << endl;
}
```

Programul anterior foloseşte 3 instrucţiuni pentru a scrie caractere în stream-ul şir de ieşire. După cum se observă, instrucţiunile de ieşire folosesc manipulatorii *setprecision* şi *ends*. Aşa cum aţi învăţat, manipulatorul *setprecision* controlează numărul de cifre afişate la dreapta punctului zecimal. Manipulatorul *ends* inserează un caracter NULL în buffer-ul stream-ului, caracter ce marchează sfârşitul şirului. Stream-ul de ieşire din programul precedent a folosit un buffer de 256 octeţi ce corespunde tabloului *şir*.

Când programele scriu date în stream-uri şir, bufferele acestor stream-uri nu pot primi un număr de caractere mai mare decât lungimea lor. Dacă un program încearcă să suprascrie un buffer, indicatorul *fail* al stream-ului este activat şi caracterele următoare sunt ignorate. Următorul program, OUT_FAIL, de exemplu, încearcă să scrie literele A până la Z într-un stream al cărui buffer are dimensiunea de 10 octeţi. Folosind funcţia membru *fail*, programul detectează eroarea şi afişează un mesaj corespunzător:

```
#include <iostream.h>
#include <strstrea.h>
#include <stdlib.h>

void main(void)
 {
   char şir[10];

   ostrstream buffer(şir, sizeof(şir));

   for (char letter = 'A'; letter <= 'Z'; letter++)
    {
      buffer << letter;

      if (buffer.fail())
       {
         cerr << endl << "Error writing the character " <<
                letter << endl;
         exit(1);
       }
      else
       cout << letter;
    }
 }
```


La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> OUT_FAIL  <ENTER>
ABCDEFGHIJ
Error writing the letter K
```

FOLOSIREA UNUI BUFFER DINAMIC

Programele anterioare au folosit un buffer de lungime fixă ce corespundea tabloului *şir*. Dacă la crearea unui stream şir de ieşire nu se specifică buffer-ul şi dimensiunea sa, atunci funcţia constructor *ostrstream* va aloca un buffer dinamic, folosind operatorul *new*. Următoarea instrucţiune, de exemplu, creează un stream şir de ieşire, denumit *buffer*, ce foloseşte un tablou dinamic:

```
ostrstream buffer;
```

Următorul program, DYNAMIC.CPP, creează un stream şir de ieşire a cărui dimensiune este dinamică. Programul scrie apoi literele alfabetului în buffer de 100 de ori (2600 de octeţi). Folosind manipulatorul *ends*, programul opreşte stream-ul şi afişează literele prin transferul caracterelor către *cout*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   ostrstream buffer;

   for (int count = 0; count < 100; count++)
     for (char letter = 'A'; letter <= 'Z'; letter++)
       buffer << letter;

   buffer << ends;

   cout << buffer.rdbuf();
 }
```

După cum se observă, programul nu specifică buffer-ul de caractere care se foloseşte cu stream-ul. În schimb, *ostrstream* va aloca memoria dinamic, din zona liberă, folosind operatorul *new*. Pentru a afişa conţinutul stream-ului, programul trebuie să apeleze buffer-ul ce conţine caracterele. În acest scop, programul foloseşte funcţia membru *rdbuf*. Fubcţia *rdbuf* returnează un pointer la începutul buffer-ului.

Când un stream de ieşire foloseşte un buffer dinamic, dimensiunea buffer-ului poate creşte (în fundal) în timpul execuţiei programului. Stream-ul va aloca volumul de memorie necesar, eventual până la epuizarea spaţiului liber de memorie. Următorul program, EAT_HEAP.CPP, creează un stream şir de ieşire

ce foloseşte un buffer dinamic. Programul umple apoi buffer-ul cu literele alfabetului până la epuizarea spaţiului de memorie liberă:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   ostrstream buffer;

   long count = 0L;

   cout << "Working...";

   while (! buffer.fail())
     for (char letter = 'A'; letter <= 'Z'; letter++, count++)
       buffer << letter;

   cout << "The number of characters buffered was " << count <<
           endl;
 }
```



CREAREA UNUI STREAM ŞIR DE IEŞIRE DINAMIC

Pentru a crea un stream şir de ieşire, trebuie declarat un obiect de tipul *ostrstream*. În mod normal, programele atribuie un buffer de caractere stream-ului, după cum se vede mai jos:

```
char °ir[256];

ostrstream buffer(°ir, sizeof(°ir));
```

Dacă programul nu specifică un buffer, funcţia constructor *ostrstream* va aloca automat zone din memoria disponibilă pentru buffer-ul stream-ului. Când programul atribuie buffer-ului dinamic un volum de caractere mai mare decât capacitatea acestuia, dimensiunea buffer-ului este mărită automat, prin alocarea de memorie din zona fragmentată. Pentru folosirea datelor din buffer-ul stream-ului, programul poate obţine un pointer la începutul buffer-ului folosind funcţia membru *rdbuf*, cum se arată mai jos:

```
cout << buffer.rdbuf();
```

FOLOSIREA FUNCŢIILOR MEMBRU ALE STREAM-ULUI ŞIR DE IEŞIRE

Câteva din programele precedente au folosit funcţiile *fail* şi *rdbuf* cu stream-urile şir de ieşire. În funcţie de cerinţele programului, la un moment dat trebuie să se lucreze cu câte un caracter al buffer-ului, sau chiar cu întregul buffer.

Următorul program, OUT_CHAR.CPP, foloseşte funcţia membru *put* pentru a plasa în buffer literele alfabetului, una câte una:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   ostrstream buffer;

   for (char letter = 'A'; letter <= 'Z'; letter++)
     buffer.put(letter);

   buffer.put((char)NULL);

   cout << buffer.rdbuf() << endl;
 }
```

Programele precedente au plasat caractere în stream-ul şir fără a lua în consideraţie numărul de caractere pe care buffer-ul îl poate conţine. Dacă programul doreşte să cunoască numărul de caractere din buffer, se poate folosi funcţia membru *pcount*. Următorul program, PCOUNT.CPP, foloseşte această funcţie pentru a afişa numărul de caractere conţinute în buffer la diferite momente ale execuţiei programului:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   ostrstream buffer;

   cout << "The buffer currently contains " << buffer.pcount() <<
           " bytes " << endl;

   buffer << "Success with C++";

   cout << "The buffer currently contains " << buffer.pcount() <<
           " bytes " << endl;

   buffer << " " << 1 << " 2 " << 3 << endl << ends;

   cout << "The buffer currently contains " << buffer.pcount() <<
           " bytes " << endl;

   cout << buffer.rdbuf() << endl;
 }
```

BLOCAREA UNUI BUFFER DINAMIC PRIN FUNCŢIA STR

La folosirea unui stream şir de ieşire dinamic, funcţia *rdbuf* este apelată pentru a obţine un pointer la buffer-ul stream-ului. Se poate, însă, folosi şi funcţia membru *str* pentru a obţine un pointer la buffer. Următorul program, USE_STR.CPP, foloseşte această funcţie pentru a afişa conţinutul buffer-ului:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   ostrstream buffer;

   buffer << "Success with C++";

   cout << buffer.str() << endl;
 }
```

La invocarea funcţiei *str*, programul blochează buffer-ul, iar acesta nu mai acceptă alte caractere. Şi iată de ce: dimensiunea unui buffer dinamic creşte în conformitate cu cerinţele programului. Pentru a creşte dimensiunea buffer-ului, funcţiile stream-ului deplasează părţi ale buffer-ului în diferite locaţii de memorie. Funcţiile stream-ului pot deplasa buffer-ul, deoarece pentru program nu are importanţă locaţia buffer-ului. Când este apelată funcţia *str*, însă, funcţiile renunţă la monopolul asupra buffer-ului, transferând programului gestiunea acestuia. Ca atare, funcţiile blochează buffer-ul, împiedicând creşterea sau deplasarea acestuia. O dată ce buffer-ul este blocat, inserarea altor caractere în buffer nu mai este permisă. În plus, când programul nu mai are nevoie de buffer, ar trebui folosit operatorul *delete* pentru eliberarea memoriei alocate pentru buffer. Din cauza acestor complexităţi, majoritatea programelor folosesc funcţia *rdbuf* în locul funcţiei *str*.

STREAM-URI ŞIR DE INTRARE/IEŞIRE

În Capitolul 3 aţi învăţat cum se deschid fişiere pentru operaţii de citire şi scriere. Analog, pot fi realizate operaţii de citire şi scriere pe stream-uri şir. Pentru a crea un stream şir capabil de a permite atât intrarea (extracţia), cât şi ieşirea (inserţia), trebuie creat un obiect de tipul *strstream*, ca mai jos:

```
strstream buffer;
```

Următorul program, INOUTSTR.CPP, creează un stream şir pentru operaţii de intrare şi ieşire. Programul trimite apoi în buffer o valoare reală, apoi introduce valoarea într-o variabilă de tip *float*:

```
#include <iostream.h>
#include <strstrea.h>
```



```
void main(void)
 {
   strstream buffer;

   float value;

   buffer << 1.2345;

   buffer >> value;

   cout << "The value is " << value << endl;
 }
```

În funcţie de cerinţele unui program, poate fi necesară introducerea unor date utilizator, prelucrarea datelor şi apoi atribuirea acestora unor variabile. De exemplu, programul ar putea mai întâi testa datele pentru a se asigura că acestea sunt numere sau text, înainte de a încerca asignarea acestor date unor variabile ale programului:

Când programele folosesc obiecte de tipul *strstream*, programul poate folosi buffere dinamice, cum s-a arătat mai înainte, sau poate specifica un buffer, după cum urmează:

```
char °ir[64];

strstream buffer(°ir, sizeof(°ir));
```

În programe pot apărea situaţii când dorim să extindem un buffer, prin adăugarea altor caractere. De exemplu, buffer-ul ar putea conţine iniţial caracterele "Hello", iar programul doreşte să completeze acest text. Pentru a adăuga text la conţinutul unui buffer se includ indicatoarele *ios::ate* sau *ios::app*, cum se arată mai jos.

Următorul program, HELLO.CPP, adaugă text la un buffer ce conţine cuvântul „Hello":

```
#include <iostream.h>
#include <strstrea.h>

void main(int argc, char **argv)
 {
   char °ir[256] = "Hello, ";

   strstream buffer(°ir, sizeof(°ir), ios::app);

   buffer << argv[1] << endl;

   cout << buffer.str();
 }
```

Pentru a afişa un mesaj folosind programul HELLO, programul trebuie apelat împreună cu mesajul dorit, astfel:

```
C:\> HELLO world! <ENTER>
Hello, world!
```

SĂ ÎNŢELEGEM RELAŢIILE DINTRE STREAM-URI

În Capitolul 11 aţi examinat relaţia dintre diferite stream-uri I/O.

Clasele de stream şir definite în fişierul antet STRSTREA.H sunt, de asemenea, legate de celelalte clase de stream. Figura 12.3, de exemplu, ilustrează modul în care stream-urile şir sunt dependente de clasele I/O discutate în Capitolul 11:

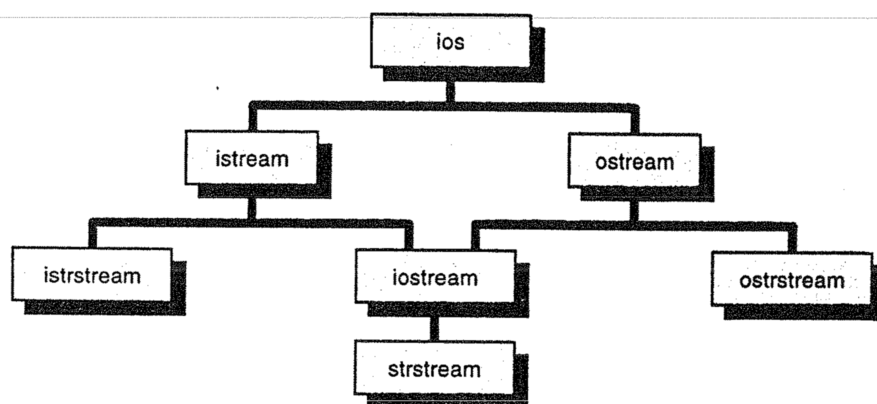


Figura 12.3 *Relaţia dintre clasele stream I/O şi clasele stream şir.*

REZUMAT

Un stream şir este o secvenţă de caractere conţinută într-un buffer de memorie şi folosită pentru conversii de date sau formatarea intrării/ieşirii. Acest capitol a examinat stream-urile şir şi operaţiile aferente în detaliu. Înainte de a trece la Capitolul 13, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Un stream şir este un buffer de caractere plasat în memorie. Folosind operatorii de stream (operatorii de inserţie şi extracţie), programele pot realiza operaţii de intrare/ieşire pe stream-uri şir.
- ✓ Fişierul antet STRSTREA.H conţine definiţiile de clase pentru stream-urile şir.
- ✓ Cele 3 clase importante pentru stream-urile şir sunt *istrsteam*, *ostrstream* şi *strstream*. Clasa *istrstream* permite realizarea operaţiilor de intrare pe stream. Clasa *ostrstream* permite efectuarea operaţiilor de ieşire, iar obiectele de tip *strstream* permit operaţii de intrare şi ieşire.
- ✓ Stream-urile şir de intrare sunt astfel denumite deoarece programele pot folosi operatorul de extracţie (>>) sau funcţiile membru ale stream-ului pentru a introduce valori din stream şi a le atribui variabilelor.
- ✓ Stream-urile şir de ieşire sunt astfel denumite deoarece programele pot folosi operatorul de inserţie (<<) sau funcţiile membru ale stream-ului pentru a extrage valori din stream.
- ✓ Deoarece stream-urile şir sunt definite de clase, ele acceptă funcţii membru. Dacă examinaţi fişierul antet STRSTREA.H, veţi găsi funcţii membru corespunzătoare funcţiilor I/O discutate în Capitolul 1.
- ✓ Când creaţi obiecte stream şir de ieşire, programele pot specifica un buffer în care stream-ul va memora datele. De asemenea, programele pot dirija stream-ul pentru a folosi un buffer dinamic, alocat de funcţiile stream-ului din zona de memorie liberă. Dacă se foloseşte buffer dinamic, dimensiunea acestuia poate creşte (în fundal) în cursul execuţiei programului.

CAPITOLUL 13
APROFUNDAREA FUNCŢIILOR VIRTUALE

În Capitolul 8 aţi învăţat cum se folosesc funcţiile virtuale în C++ pentru a realiza polimorfismul. Tot în Capitolul 8 aţi citit că funcţiile virtuale sunt posibile datorită operaţiilor „discrete" efectuate de compilator. Acest capitol examinează câteva din aceste operaţii în încercarea de a clarifica, în parte, câteva din enigmele ce însoţesc funcţiile virtuale. Când veţi termina acest capitol, veţi cunoaşte următoarele:

- Ce sunt legăturile statice şi dinamice
- Ce este o tabelă de funcţii virtuale
- Preţul plătit pentru flexibilitatea sporită generată de funcţiile virtuale

ANALIZA UNUI EXEMPLU SIMPLU

Aşa cum aţi învăţat în Capitolul 8, polimorfismul permite unui obiect să ia două sau mai multe forme. Următorul program, SHAPES.CPP, foloseşte funcţii virtuale pentru a crea un pointer la obiectul polimorfic *shape*. Începem cu crearea următoarei clase de bază *shape*:

```
class shape {
  public:
    shape(char *name, int x, int y);
    virtual void draw(void);
  protected:
    char name[64];
    int x;
    int y;
};
```

Se observă că programul declară funcţia *draw* ca fiind virtuală. Programul derivează apoi obiectele *circle* şi *square*, astfel:

```
class circle : public shape {
  public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
  private:
    char name[64];
    int radius;
};
```

```
class square : public shape {
  public:
    square(char *name, int x, int y, int side);
  private:
    char name[64];
    int side;
};
```

Folosind funcţiile virtuale, programul va atribui pointerului obiect *graphic* diferite forme (figuri geometrice). Următoarele instrucţiuni implementează programul SHAPES.CPP:

```
#include <iostream.h>
#include <string.h>

class shape {
  public:
    shape(char *name, int x, int y);
    virtual void draw(void);
  protected:
    char name[64];
    int x;
    int y;
};

class circle : public shape {
  public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
  private:
    char name[64];
    int radius;
};

class square : public shape {
  public:
    square(char *name, int x, int y, int side);
  private:
    char name[64];
    int side;
};

shape::shape(char *name, int x, int y)
 {
   strcpy(shape::name, name);
   shape::x = x;
   shape::y = y;
 }

void shape::draw(void)
 {
```



```
   cout << "Drawing the shape: " << name << endl;
 }

circle::circle(char *name, int x, int y, int radius) :
   shape(name, x, y)
 {
   circle::radius = radius;
 }

void circle::draw(void)
 {
   cout << "Circles are being drawn: ";
   cout << "x " << x << " y " << y << " radius " << radius << endl;
 }

square::square(char *name, int x, int y, int side) :
   shape(name, x, y)
 {
   square::side = side;
 }

void main(void)
 {
   circle ball("Circle", 10, 20, 30);

   square box("Square", 40, 50, 40);

   shape *graphic = &ball;

   graphic->draw();

   graphic = &box;

   graphic->draw();
 }
```

Aşa cum se observă, pointerul *graphic* ia două forme, *circle* şi *square*.

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SHAPES <ENTER>
Circles are being drawn: x 10 y 20 radius 30
Drawing the shape: Square
```

POLIMORFISMUL NU ÎNSEAMNĂ SUPRAPUNEREA FUNCŢIILOR

Aşa cum aţi învăţat în Capitolul 8, polimorfismul nu este acelaşi lucru cu suprapunerea funcţiilor. De exemplu, următorul program, SUMARRAY.CPP, suprapune funcţia *sum_array*:

```
#include <iostream.h>

float sum_array(float *array, int num_elements)
 {
   float sum = 0.0;

   for (int i = 0; i < num_elements; i++)
     sum += array[i];

   return(sum);
 }

long sum_array(int *array, int num_elements)
 {
   long sum = 0L;

   for (int i = 0; i < num_elements; i++)
     sum += array[i];

   return(sum);
 }

void main(void)
 {
   int int_array[5] = {1, 2, 3, 4, 5};
   float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

   cout << "Values in int array: " << sum_array(int_array, 5) <<
           endl;
   cout << "Values in float array: " << sum_array(float_array, 5) <<
           endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> SUMARRAY <ENTER>
Values in int array: 15
Values in float array: 16.5
```

În timpul compilării, compilatorul C++ determină care din funcţii va fi folosită, bazându-se pe parametrii efectivi transferaţi funcţiilor sau pe tipul valorilor returnate de aceste funcţii. Compilatorul va substitui atunci funcţia corectă pentru fiecare apelare. Astfel de substituţii efectuate la compilare se numesc *legături statice*. Ele sunt efectuate *înainte* de execuţia programului. Când într-un program se realizează polimorfism, pe de altă parte, substituţiile sunt efectuate *în timpul* execuţiei programului. Astfel de substituţii la rulare se numesc *legături dinamice*.

SĂ ÎNŢELEGEM LEGĂTURILE DINAMICE

Din punctul de vedere al compilatorului, suprapunerea funcţiilor este o sarcină uşoară, deoarece acesta cunoaşte modul de utilizare a fiecărui program înainte de lansarea în execuţie. În cazul următorului program, însă, obiectul care este atribuit pointerului obiect *graphic* nu este cunoscut anterior execuţiei programului. Programul ASKSHAPE.CPP cere utilizatorului să selecteze obiectul *circle* sau *square*. În funcţie de selecţia utilizatorului, programul va atribui pointerului unul din obiecte:

```
#include <ctype.h>

void main(void)
 {
   circle ball("Circle", 10, 20, 30);

   square box("Square", 40, 50, 40);

   shape *graphic;

   char choice;

   do {
     cout << "Type C for circle S for square: ";
     cin >> choice;
     choice = toupper(choice);

     if (choice == 'S')
       graphic = &box;
     else if (choice == 'C')
       graphic = &ball;
   } while ((choice != 'C') && (choice != 'S'));

   cout << endl << endl;

   graphic->draw();
 }
```

Observaţie: *Programul ASKSHAPE.CPP nu a inclus şi declaraţiile de clasă şi funcţiile membru.*

Aşa cum se poate observa, în funcţie de opţiunea utilizatorului, programul atribuie diferiţi pointeri obiect la pointerul *graphic*. Cu alte cuvinte, programul trebuie să determine adresa corectă a obiectului în timpul execuţiei. Pentru a determina astfel de adrese, programul foloseşte intrinsec o *tabelă de funcţii virtuale*. Această tabelă este construită de compilator.

SĂ ÎNŢELEGEM TABELELE DE FUNCŢII VIRTUALE

O tabelă de funcţii virtuale este o tabelă pe care C++ o foloseşte la execuţia programului pentru a localiza funcţia membru corectă, în cazul utilizării polimorfismului. Tabela conţine rubrici pentru obiectele ce folosesc clase virtuale. Când programul foloseşte o astfel de funcţie, C++ caută în tabelă adresa funcţiei corecte, apoi apelează funcţia.

De fiecare dată când o clasă foloseşte una sau mai multe funcţii virtuale, compilatorul C++ îi adaugă un pointer special ce indică tabela virtuală. Următorul program, TABLEPTR.CPP, foloseşte operatorul *sizeof* pentru a afişa dimensiunea a două clase foarte asemănătoare. Singura diferenţă între cele două clase este aceea că prima dintre ele foloseşte o funcţie virtuală. Ca urmare, prima clasă conţine un pointer special la tabela de funcţii virtuale:

```
#include <iostream.h>

class shape1 {
  public:
    shape(char *name, int x, int y);
    virtual void draw(void);
  protected:
    char name[64];
    int x;
    int y;
};

class shape2 {
  public:
    shape(char *name, int x, int y);
    void draw(void);
  protected:
    char name[64];
    int x;
    int y;
};

void main(void)
 {
   cout << "Size of virtual class: " << sizeof(shape2) << endl;
   cout << "Size of nonvirtual class: " << sizeof(shape1) << endl;
 }
```

La compilarea şi execuţia acestui program în Borland C++, pe ecran va apărea:

```
C:\> TABLEPTR  <ENTER>
Size of virtual class: 68
Size of nonvirtual class: 70
```

După cum se observă, clasa care foloseşte funcţia virtuală este mai cuprinzătoare, datorită pointerului la tabela virtuală.

SUPRASOLICITAREA GENERATĂ DE FUNCŢIILE VIRTUALE

Deşi funcţiile virtuale aduc o mare flexibilitate în programe, folosirea lor nu se face fără un anumit efort de timp şi de memorie. De fiecare dată când programul foloseşte un obiect ce conţine una sau mai multe funcţii virtuale, este folosit un pointer special pentru accesul la tabela virtuală, care va indica, la rândul ei, adresa funcţiei corecte. Deoarece apelarea unei funcţii virtuale cere o indirectare suplimentară a memoriei, ea se va derula mai încet decât apelarea unei funcţii standard. Următorul program, OVERHEAD.CPP, apelează două funcţii similare de 5 milioane de ori, măsurând timpul necesar apelării fiecărei funcţii. Prima funcţie este virtuală, în timp ce a doua este o funcţie standard:

```
#include <iostream.h>
#include <string.h>
#include <time.h>

class shape {
  public:
    shape(int x) { shape::x = x; };
    virtual void draw(void);
  protected:
    int x;
};

class circle {
  public:
    circle(int x) { circle::x = x; };
    void draw(void);
  private:
    int x;
};

void shape::draw(void)
 {
   return ;
 }

void circle::draw(void)
 {
   return ;
 }

void main(void)
 {
   time_t start_time, stop_time;

   circle ball(1);

   shape box(2);
```

```
  long int i;

  cout << "Working..." << endl;

  time(&start_time);
  for (i = 0; i < 5000000L; i++)
    box.draw();
  time(&stop_time);

  cout << "Virtual required: " << stop_time - start_time << endl;

  time(&start_time);
  for (i = 0; i < 5000000L; i++)
    ball.draw();
  time(&stop_time);

  cout << "Standard required: " << stop_time - start_time << endl;
}
```

În funcţie de calculatorul folosit, timpul necesar fiecărei funcţii poate diferi. Totuşi, dată fiind viteza foarte mare a procesoarelor de astăzi, consumul suplimentar asociat cu funcţiile virtuale devine minimal pentru majoritatea aplicaţiilor şi poate fi neglijat.

SĂ ÎNŢELEGEM DESTRUCTORII VIRTUALI

În Capitolul 2 s-au prezentat funcţiile constructor şi destructor. Toate programele din această carte au făcut uz de funcţiile constructor, iar unele au folosit şi funcţiile destructor. Când definiţi funcţii constructor şi destructor, reţineţi următoarele: *funcţiile constructor nu pot fi virtuale, dar funcţiile destructor pot fi virtuale.*

Aşa cum aţi învăţat, un obiect polimorfic poate lua una sau mai multe forme. Când obiectul corespunzător este distrus, programul trebuie să apeleze destructorul corect. De exemplu, să considerăm următorul program, BADDESTR.CPP, care foloseşte pointerul polimorfic *graphic*. Programul defineşte o funcţie destructor pentru fiecare tip de obiect, care afişează un mesaj ce informează utilizatorul despre apelarea destructorului.

```
#include <iostream.h>
#include <string.h>

class shape {
  public:
    shape(char *name, int x, int y);
    virtual void draw(void);
    ~shape(void);
  protected:
    char name[64];
```



```
    int x;
    int y;
};

class circle : public shape {
  public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
    ~circle(void);
  private:
    char name[64];
    int radius;
};

shape::shape(char *name, int x, int y)
{
  strcpy(shape::name, name);
  shape::x = x;
  shape::y = y;
}

void shape::draw(void)
{
  cout << "Drawing the shape: " << name << endl;
}

shape::~shape()
{
  cout << "In shape's destructor" << endl;
}

circle::circle(char *name, int x, int y, int radius) :
  shape(name, x, y)
{
  circle::radius = radius;
}

void circle::draw(void)
{
  cout << "Circles are being drawn: ";
  cout << "x " << x << " y " << y << " radius " << radius << endl;
}

circle::~circle(void)
{
  cout << "In circle's destructor" << endl;
}

void main(void)
```

```
{
  shape *graphic = new circle("Circle", 10, 20, 30);

  graphic->draw();
  delete graphic;
}
```

Aşa cum se observă, programul creează un obiect *circle*, atribuind un pointer al obiectului către variabila polimorfică *graphic*. După ce programul desenează obiectul, se foloseşte operatorul *delete* pentru distrugerea obiectului. Din păcate, deoarece *graphic* este un pointer la clasa *shape*, programul invocă destructorul clasei *shape*, şi nu destructorul clasei *circle*. Drept rezultat, când compilaţi şi executaţi acest program, pe ecran va apărea:

```
C:\> BADDESTR <ENTER>
Circles are being drawn: x 10 y 20 radius 30
In shape's destructor
```

În continuare, să presupunem că clasa *circle* a alocat memorie care trebuie disponibilizată de către o funcţie destructor. După cum aţi văzut, programul nu apelează destructorul clasei *circle*.

Totuşi, dacă se transformă funcţia destructor a clasei *shape* într-o funcţie virtuală, programul va apela, consecutiv, constructorii claselor *circle* şi *shape*, cum se arată mai jos:

```
C:\> BADDESTR <ENTER>
Circles are being drawn: x 10 y 20 radius 30
In circle's destructor
In shape's destructor
```

După cum se observă, programul apelează corect fiecare din funcţiile destructor. Când un program foloseşte polimorfismul şi diferite clase alocă memorie, se recomandă folosirea funcţiilor destructor virtuale pentru eliberarea corectă a memoriei.

Rezumat

În Capitolul 8 aţi examinat polimorfismul şi funcţiile virtuale în detaliu. În acest capitol aţi învăţat câteva din operaţiile intrinseci pe care le efectuează C++ pentru a implementa funcţiile virtuale. Înainte de a trece la Capitolul 15, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Legăturile statice sunt substituţii efectuate de compilatorul C++ înainte de execuţia programului, pentru a determina ce funcţii trebuie apelate.
- ✓ Legăturile dinamice sunt substituţii făcute în momentul execuţiei programului, când acesta realizează polimorfismul.

- ✓ O tabelă de funcţii virtuale conţine rubrici pentru obiecte ce folosesc clase virtuale.
- ✓ Deoarece apelarea unei funcţii virtuale necesită o indirectare suplimentară a memoriei pentru a avea acces la tabela de funcţii virtuale, aceasta se derulează mai încet decât apelarea unei funcţii standard.
- ✓ Când într-un program se realizează polimorfismul şi clasele acestuia alocă memoria dinamic, trebuie folosite funcţii virtuale pentru eliberarea corectă a memoriei.

CE SUNT CAZURILE DE EXCEPŢIE

O excepţie este o eroare definită de program. Dacă o funcţie lucrează, de exemplu, cu tablouri, o excepţie ar putea apărea dacă programul transferă funcţiei un index ce depăşeşte limitele tabloului. Când testează valoarea indexului, funcţia poate genera sau *lansa* o excepţie. Analog, dacă o funcţie adună două valori întregi, se poate genera o excepţie dacă rezultatul depăşeşte domeniul de valori întregi. Singurele excepţii care apar în cadrul unui program sunt cele definite de program sau de bibliotecile folosite de program. Evenimentele hard, precum defectul de pagină, eroare la citirea pe disc sau alte întreruperi, nu generează excepţii.

Pentru a realiza tratarea erorilor, se folosesc în programe cuvintele cheie *try*, *throw* şi *catch*. Să presupunem, de exemplu, că programul foloseşte funcţia *fill_array* pentru a citi valori dintr-un fişier şi a le insera într-un tablou de dimensiune fixă. Dacă fişierul conţine mai multe valori decât poate memora tabloul, funcţia ar trebui să genereze o excepţie.

Înainte să apeleze funcţia, programul foloseşte instrucţiunea *try* pentru a permite tratarea excepţiilor:

```
try {
    fill_array(some_array, 100);
}
```

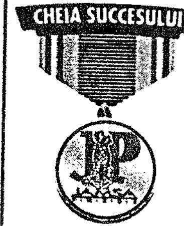
După ce programul activează tratarea excepţiilor, se poate încerca detecţia anumitor excepţii folosind instrucţiunea *catch*, cum se indică mai jos:

```
try {
    fill_array(some_array, 100);
}

catch (array_overflow) {
     cerr << "Too many values for array to handle" << end;
     exit(1);
}

catch (file_not_found) {
     cerr << "Could not find the data file" << endl;
     exit(1);
}
```

În exemplul precedent, programul apelează funcţia *fill_array* în cadrul instrucţiunii *try*. Apoi, folosind două instrucţiuni *catch*, programul va încerca să detecteze posibila apariţie a două erori. Dacă apare oricare din cele două excepţii, programul va efectua instrucţiunile corespunzătoare (rutina specifică de tratare a excepţiei). Numai o singură excepţie poate apărea la un moment dat. Cu alte cuvinte, programul anterior nu poate genera în acelaşi timp excepţiile *array_overflow* şi *file_not_found*.



PRELUAREA EXCEPŢIILOR ÎNTR-UN PROGRAM

Pentru a detecta şi manipula o excepţie într-un program, trebuie folosite instrucţiunile *try* şi *catch*. Pentru a activa tratarea excepţiilor în cazul unei anumite secvenţe de operaţii, se foloseşte instrucţiunea *try*:

```
try {
   some_operation(a, b, c);
}
```

Pentru a determina apoi dacă o excepţie a avut loc, se foloseşte instrucţiunea *catch*:

```
catch (exception_1) {
   // instructiuni
}

catch (exception_2) {
   // instructiuni
}
```

CUM SE DENUMESC EXCEPŢIILE

Fiecare excepţie folosită într-un program are nume unic. Excepţiile se denumesc prin crearea claselor de excepţii. De exemplu, următoarele instrucţiuni definesc 3 clase de excepţii:

```
class open_error { };
class read_error { };
class write_error { };
```

În acest caz, clasele de excepţii nu au membri. În exemplele următoare, veţi vedea cum variabilele membru permit claselor de excepţii să returneze mai multe informaţii despre excepţie. Pentru a genera o excepţie, programul trebuie să o lanseze cu ajutorul instrucţiunii *throw*. De exemplu, următoarea instrucţiune lansează excepţia *write_error*:

```
throw write_error();  // Include parenthesis after the class name
```

În acest mod, excepţiile sunt lansate şi ulterior preluate de rutina de tratare folosind instrucţiunea *catch*. Următorul program, COPY_EXP.CPP, foloseşte excepţiile pentru a determina apărute la deschiderea, citirea şi scrierea fişierelor. Dacă o astfel de eroare apare în funcţia *perform_copy*, funcţia va lansa excepţia corespunzătoare. Folosind instrucţiunea *catch*, programul preia fiecare tip de excepţie:

```
#include <fstream.h>
#include <stdlib.h>
```

```
class open_error { };
class read_error { };      ——— Definirea claselor de excepţii
class write_error { };

void perform_copy(char *source_file, char *target_file)
 {
   ifstream source(source_file, ios::binary);

   char line[1];                          Generarea unei excepţii

   if (source.fail())
     throw open_error();
   else
    {
      ofstream target(target_file, ios::binary);

      if (target.fail())
        throw open_error();
      else
        {
          while (! source.eof() && ! source.fail())
            {
              source.read(line, sizeof(line));

              if (source.good())
               {
                 target.write(line, sizeof(line));

                 if (target.fail())
                   throw write_error();
               }
              else if (! source.eof())
                throw read_error();
            }
          source.close();
          target.close();
        }
    }
 }

void main(int argc, char **argv)
 {
   try {
     perform_copy(argv[1], argv[2]);
   }

   catch (open_error) {       ——— Preluarea unei excepţii
     cerr << "Error opening file" << endl;
     exit(1);
   }
```



```
     catch (read_error) {
     cerr << "Error reading file" << endl;
     exit(1);
   }                                      Preluarea unei excepţii

   catch (write_error) {
     cerr << "Error writing file" << endl;
     exit(1);
   }
 }
```



DENUMIREA ŞI GENERAREA EXCEPŢIILOR

Fiecare excepţie dintr-un program are nume unic. O excepţie se defineşte prin crearea unei clase de excepţii de forma:

```
class out_of_memory { };
```

Pentru a activa ulterior excepţia, programul foloseşte cuvântul cheie *throw* şi numele excepţiei, cum se arată mai jos:

```
throw out_of_memory();
```

Observaţi parantezele ce urmează după numele excepţiei în instrucţiunea *throw*. În acest caz, excepţia va fi tratată de codul ce preia excepţia *out_of_memory*:

```
try {
    some_operation()
}
catch (out_of_memory) {
  // Statements
}
```

EXCEPŢII LOCALE ALE UNEI CLASE

În exemplul anterior, clasele de excepţii au fost globale pentru întregul program. În multe alte situaţii, însă, excepţiile sunt locale unei clase. De exemplu, să presupunem că un program foloseşte o clasă tablou pentru a memora valori întregi:

```
class Array {
  public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);

  private:
    int *buffer;
```

```
  int size;
};
```

Când în program se creează un obiect de tip *Array*, trebuie specificată o dimensiune a bufferului, astfel:

```
Array vector(100);
```

În acest caz, programul va aloca un buffer capabil de a memora 100 de valori întregi. Funcţia *fill_array* va permite programului să atribuie valori consecutive elementelor tabloului. De exemplu, următoarea instrucţiune atribuie valorile 1 până la 25 primelor 25 de elemente (restul elementelor rămâne neschimbat):

```
vector.fill_array(25);
```

În mod similar, funcţia *show_array* permite afişarea unui anumit număr de elemente ale tabloului. Următoarea instrucţiune, de exemplu, afişează valorile primelor 10 elemente:

```
vector.show_array(10);
```

Următorul program, ARRAY.CPP, foloseşte clasa *Array* pentru a crea un vector de valori întregi:

```
#include <iostream.h>

class Array {
  public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);

  private:
    int *buffer;
    int size;
};

Array::Array(int size)
 {
   buffer = new int[size];
   Array::size = size;
 }

void Array::fill_array(int num_values)
 {
   for (int i = 0; i < num_values; i++)
     buffer[i] = i;
 }

void Array::show_array(int num_values)
 {
```



```
   for (int i = 0; i < num_values; i++)
     cout << buffer[i] << endl;
 }

void main(void)
 {
   Array vector(200);

   vector.fill_array(10);

   vector.show_array(10);
 }
```

După cum se observă, programul foloseşte tabloul fără a testa valorile din indexul transmis funcţiilor *fill_array* şi *show_array*. Să presupunem, de exemplu, că programul apelează funcţia *fill_array* cu valoarea de index 500:

```
vector.file_array(500);
```

În acest caz, programul va atribui tabloului valorile de la 1 la 500, depăşind tabloul cu 300 de valori. O variantă mai bună a programului ar trebui să testeze valoarea indexului, pentru a se asigura că aceasta este corectă, înainte de atribuirea valorilor întregi.

Următorul program, ARRAYEXC.CPP, foloseşte tratarea excepţiilor pentru a semnaliza o operaţie ce poate depăşi limitele tabloului. Deoarece excepţiile sunt specifice clasei *Array*, domeniul excepţiilor este limitat la această clasă, definind excepţiile ca membri ai clasei:

```
#include <iostream.h>
#include <stdlib.h>

class Array {
  public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);
    class range { };
  private:
    int *buffer;
    int size;
};

Array::Array(int size)
 {
   buffer = new int[size];
   Array::size = size;
 }

void Array::fill_array(int num_values)
 {
```

```
  if (num_values > size)
    throw range();

  for (int i = 0; i < num_values; i++)
    buffer[i] = i;
}

void Array::show_array(int num_values)
{
  if (num_values > size)
    throw range();

  for (int i = 0; i < num_values; i++)
    cout << buffer[i] << endl;
}

void main(void)
{
  Array vector(200);

  try {
    vector.fill_array(10);
  }
  catch (Array::range) {
    cerr << "Invalid array index in fill_array" << endl;
    exit(1);
  }

  try {
    vector.show_array(500);
  }
  catch (Array::range) {
    cerr << "Invalid array index in show_array" << endl;
    exit(1);
  }
}
```

După cum se observă, programul apelează funcţia *show_array* cu o valoare incorectă a indexului, şi anume 500. Când funcţia va examina această valoare, ea va lansa excepţia, care este preluată de instrucţiunea *catch*.

TRATAREA EXCEPŢIILOR

Când apare o excepţie, programul trebuie să determine modul în care aceasta va fi tratată. În cazul programelor precedente se afişau numai mesaje de erori, urmate de încheierea programului. Alte programe pot încerca să rezolve erorile. Să presupunem, de exemplu, că un program primeşte o excepţie de tip memorie insuficientă. În loc de a-şi încheia execuţia, programul ar putea elibera memoria ocupată sau ar putea colecta resturile de memorie fragmentată, cum se

va discuta în Capitolul 15. În astfel de cazuri, rutina de tratare va avea nevoie de acces la pointerii a căror memorie poate fi disponibilizată sau funcţia trebuie să fie suficient de complicată pentru a traversa zona de memorie fragmentată, disponibilizând locaţiile neesenţiale. După ce memoria cerută de program devine disponibilă, programul poate repeta operaţia care a cauzat eroarea.

În alte situaţii, se poate dori pur şi simplu terminarea programului. Dacă un program generează o excepţie şi aceasta nu este preluată, programul va lansa o rutină de tratare prestabilită care va încheia în mod automat programul. Următorul program, BAD_EXC.CPP, lansează excepţia *no_such_device*. Deoarece nu există cod care să preia excepţia, programul se va termina pur şi simplu:

```
#include <iostream.h>

class no_such_device { };     // Exception class
class file_not_found { };

void show_files(char disk_drive)
{
  if ((disk_drive < 'A') || (disk_drive > 'Z'))
    throw no_such_device();
  else
    cout << "Would display files here" << endl;
}

void main(void)
{
  try {
   show_files('a');
  }
  catch (file_not_found) {
    cerr << "Error finding file" << endl;
  }
}
```

În cazul precedent, programul detectează excepţia *file_not_found*, nu şi excepţia *no_such_device*. Deoarece excepţia lansată nu este preluată, programul se încheie.

INFORMAŢII SUPLIMENTARE DESPRE O EXCEPŢIE

Când se generează excepţii, uneori este necesară includerea unor informaţii suplimentare despre cauza erorilor. De exemplu, programul COPY_EXP.CPP, prezentat anterior, lansează excepţia *open_error* când nu poate deschide fişierul sursă sau fişierul destinaţie. Din păcate, codul de program care preia excepţia nu poate determina care din cele două fişiere nu a putut fi deschis. O soluţie mai bună ar fi să se returneze o valoare ce poate fi testată de program, cum ar fi valoarea 0 pentru fişierul sursă şi valoarea 1 pentru fişierul destinaţie.

O astfel de facilitate poate fi realizată uşor prin adăugarea unei variabile membru la clasa de excepţii care să memoreze valoarea respectivă. În al doilea rând, trebuie creată o funcţie membru a clasei ce poate fi lansată cu valoarea corespunzătoare. Următoarele instrucţiuni ilustrează modul în care se poate modifica clasa de excepţii *open_file*:

```
class open_error {
 public:
   int value;
   open_error(int value) { open_error::value = value; };
};
```

După cum se poate observa, clasa conţine variabila membru *value* şi o funcţie constructor ce atribuie acesteia valoarea specificată. La lansarea excepţiei, programul poate specifica o valoare drept parametru, cum se arată mai jos:

```
throw open_error(1);  // Error opening target file
```

În acest caz, instrucţiunea *throw* determină crearea unui obiect de tip excepţie, iar la execuţia funcţiei constructor a obiectului, valoarea parametrului va fi atribuită variabilei membru. Când programul preia excepţia, el primeşte de fapt obiectul eroare şi prin urmare poate examina variabilele membru, cum se indică în continuare:

```
catch (open_error exception) {
   if (exception.value)
    cerr << "Error opening target file" << endl;
  else
    cerr << "Error opening source file" << endl;
  exit(1);
}
```

Pentru a se referi la obiectul *open_error*, programul foloseşte o variabilă denumită *exception*. Testând valoarea variabilei membru *value*, programul poate determina dacă eroarea este cauzată de fişierul sursă sau fişierul destinaţie. Următorul program, EXPVALUE.CPP, foloseşte o variabilă membru pentru a determina care fişier cauzează eroarea:

```
#include <fstream.h>
#include <stdlib.h>

class open_error {
  public:
    int value;
    open_error(int value) { open_error::value = value; };
};

class read_error { };
class write_error { };
```



```
void perform_copy(char *source_file, char *target_file)
 {
   ifstream source(source_file, ios::binary);

   char line[1];

   if (source.fail())
     throw open_error(0);
   else
    {
      ofstream target(target_file, ios::binary);

      if (target.fail())
        throw open_error(1);
      else
        {
          while (! source.eof() && ! source.fail())
            {
              source.read(line, sizeof(line));

              if (source.good())
               {
                 target.write(line, sizeof(line));

                 if (target.fail())
                   throw write_error();
               }
              else if (! source.eof())
                throw read_error();
          }
        source.close();
        target.close();
      }
  }
}

void main(int argc, char **argv)
 {
   try {
     perform_copy(argv[1], argv[2]);
   }

   catch (open_error exception) {
     if (exception.value)
       cerr << "Error opening target file" << endl;
     else
       cerr << "Error opening source file" << endl;
     exit(1);
   }
```

```
  catch (read_error) {
    cerr << "Error reading file" << endl;
    exit(1);
  }

  catch (write_error) {
    cerr << "Error writing file" << endl;
    exit(1);
  }
}
```

În mod analog, următorul program, EXSTRING.CPP, modifică programul precedent prin includerea în clasa de excepţii a numelui fişierului sub formă de şir de caractere:

```
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

class open_error {
  public:
    int value;
    char filename[64];
    open_error(int value) {
       open_error::value = value;
       filename[0] = NULL;
    };
    open_error(int value, char *filename) {
       open_error::value = value;
       strcpy(open_error::filename, filename);
    };
};

class read_error { };
class write_error { };

void perform_copy(char *source_file, char *target_file)
 {
   ifstream source(source_file, ios::binary);

   char line[1];

   if (source.fail())
     throw open_error(0, source_file);
   else
    {
      ofstream target(target_file, ios::binary);

      if (target.fail())
        throw open_error(1, target_file);
      else
```



```
       {
         while (! source.eof() && ! source.fail())
           {
             source.read(line, sizeof(line));

             if (source.good())
              {
                target.write(line, sizeof(line));

                if (target.fail())
                  throw write_error();
              }
             else if (! source.eof())
                throw read_error();
           }
         source.close();
         target.close();
       }
    }
 }

void main(int argc, char **argv)
 {
   try {
     perform_copy(argv[1], argv[2]);
   }

   catch (open_error exception) {
     if (exception.value)
       {
         if (exception.filename)
           cerr << "Error opening the target file " <<
                   exception.filename << endl;
         else
           cerr << "Error opening target file" << endl;
       }
     else
       {
         if (exception.filename)
           cerr << "Error opening the source file " <<
                   exception.filename << endl;
         else
           cerr << "Error opening source file" << endl;
       }

     exit(1);
   }

   catch (read_error) {
     cerr << "Error reading file" << endl;
     exit(1);
```

```
  }

  catch (write_error) {
    cerr << "Error writing file" << endl;
    exit(1);
  }
}
```



RETURNAREA UNOR VALORI ALĂTURI DE EXCEPŢIE

La generarea excepţiilor în programe, se pot specifica informaţii suplimentare despre excepţii. Acest lucru se poate realiza prin includerea unor variabile membru în clasa de excepţii şi a unor funcţii constructor ce pot atribui acestor variabile valorile unor parametri. Când programul preia excepţia, el poate examina variabilele membru ale excepţiei pentru a găsi mai multe informaţii despre cauza acesteia.

PRELUAREA MAI MULTOR EXCEPŢII ÎNTR-O SINGURĂ INSTRUCŢIUNE CATCH

Aşa cum aţi învăţat, dacă programul nu preia o anumită excepţie, se va apela o rutină prestabilită şi programul se va încheia. Drept urmare, programele pot avea o serie de instrucţiuni *catch*. În funcţie de modul de tratare al excepţiilor, rutinele de tratare ar putea fi similare celor prezentate mai jos:

```
catch (file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (source_file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (target_file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (temp_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
```

După cum se poate observa, cele 4 rutine de tratare a excepţiilor realizează operaţii similare. În aceste situaţii, când excepţiile sunt oarecum asemănătoare, s-ar putea efectua o grupare cu ajutorul unui tip de enumerare, astfel:

```
enum FileException { Open, Source, Target, Temp };
```


Apoi, în program, erorile se pot lansa astfel:

```
throw FileException(Source);
```

Când programul preia excepţia, se poate detecta valoarea corespunzătoare folosind o instrucţiune *switch*. Următorul program, COMBINE.CPP, foloseşte o instrucţiune *switch* pentru a determina excepţia lansată:

```
#include <iostream.h>
#include <stdlib.h>

enum FileException { Open, Source, Target, Temp };

void some_operation(void)
{
   throw FileException(1);
}

void main(void)
{
   try {
     some_operation();
   }
   catch (FileException value) {
     switch (value) {
       case Open: cerr << "Open file error" << endl;
                  break;
       case Source: cerr << "Source file error" << endl;
                  break;
       case Target: cerr << "Target file error" << endl;
                  break;
       case Temp: cerr << "Temporary file error" << endl;
                  break;
     };
     exit(1);
   }
}
```

După cum se observă, programul preia excepţia şi creează o variabilă denumită *value*. Folosind o instrucţiune SWITCH, programul poate determina tipul excepţiei survenite. Exersaţi cu acest program schimbând valoarea excepţiei lansate şi urmăriţi cum programul îşi schimbă rezultatele.

CE SE ÎNTÂMPLĂ LA LANSAREA UNEI EXCEPŢII

Când o funcţie generează o excepţie, ea nu îşi mai continuă execuţia, ca şi când s-ar fi executat o instrucţiune *return*. Apoi, lanţul de apel al funcţiilor este întrerupt până la găsirea unei rutine de tratare a acelei excepţii. Într-o rutină de tratare, activată printr-o instrucţiune *catch*, execuţia continuă cu acea rutină. De

exemplu, următorul program, EXPCHAIN.CPP, apelează o serie de funcţii până când funcţia *three* lansează o excepţie. Excepţia este preluată de *main*:

```
#include <iostream.h>

class Exception { };

void three(void)
 {
   cout << "In three, about to throw exception" << endl;
   throw Exception();
 }

void two(void)
 {
   cout << "In two, about to call three" << endl;
   three();
   cout << "Back in two" << endl;
 }

void one(void)
 {
   cout << "In one, about to call two" << endl;
   two();
   cout << "Back in one" << endl;
 }

void main(void)
 {
   try {
     one();
   }
   catch (Exception) {
     cout << "Caught the exception in main" << endl;
   }

   cout << "Hello, world" << endl;
 }
```

Când C++ întrerupe lanţul de apeluri în căutarea unei funcţii ce tratează excepţia, instrucţiunile din funcţiile întrerupte nu mai sunt executate. De exemplu, dacă compilaţi şi executaţi acest program, pe ecran va apărea:

```
C:\> EXPCHAIN <ENTER>
In one, about to call two
In two, about to call three
In three, about to throw exception
Caught the exception in main
Hello, world
```


După cum se observă, câteva instrucţiuni din fiecare din funcţiile anterioare nu se mai execută. Exersaţi cu acest program, incluzând o rutină de tratare în funcţia *two*, astfel:

```
void two(void)
 {
   cout << "In two, about to call three" << endl;
   try {
     three();
   }
   catch (Exception) {
     cout << "Caught the exception in two" << endl;
   }
   cout << "Back in two" << endl;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> EXPCHAIN <ENTER>
In one, about to call two
In two, about to call three
In three, about to throw exception
Caught the exception in two
Back in two
Back in one
Hello, world
```

Când excepţia este preluată de funcţia *two*, aceasta îşi va continua execuţia, va ceda controlul funcţiei *one*, care, la rândul ei, va ceda controlul funcţiei *main*. Deoarece excepţia a fost tratată de funcţia *two*, ea nu va mai fi activă când controlul revine funcţiei *main*, de aceea va fi executată instrucţiunea *catch*.



EXCEPŢIILE ÎNTRERUP LANŢUL DE APEL AL FUNCŢIILOR

Când o funcţie lansează o excepţie, execuţia funcţiei este oprită, ca şi cum funcţia ar fi executat o instrucţiune *return*. C++ caută apoi în lanţul de apel o funcţie de tratare a excepţiei. Dacă funcţia următoare din lanţul de apel nu tratează excepţia, execuţia acelei funcţii este imediat oprită. Dacă o funcţie a creat un obiect local, este apelată funcţia destructor a obiectului înainte ca funcţia să îşi încheie execuţia.

ATENŢIE LA EXCEPŢIILE SUBPROGRAMELOR DE BIBLIOTECĂ

Pe măsură ce folosirea excepţiilor ia amploare, veţi întâlni funcţii de bibliotecă şi funcţii executate la rulare ce generează excepţii. Când citiţi documentaţia

acestor funcţii, notaţi, posibilele excepţii pe care acestea le pot lansa. Dacă programul nu preia astfel de excepţii, atunci va fi apelată rutina de tratare prestabilită, care va încheia execuţia programului în majoritatea situaţiilor.

SPECIFICAREA INTERFEŢEI PENTRU EXCEPŢII

Aşa cum se ştie, prototipul unei funcţii specifică tipul de valori returnat de funcţie şi tipul parametrilor. Când aveţi funcţii ce generează excepţii, se pot specifica posibilele excepţii printr-o instrucţiune similară unui prototip de funcţie. De exemplu, să presupunem că funcţia *perform_copy* poate genera excepţiile *open_error, read_error* şi *write_error*. Prototipul funcţiei va putea fi specificat astfel:

```
void perform_copy(char *source_file, char *target_file)
  throw (open_error, read_error, write_error);
```

La specificarea în acest mod a excepţiilor unei funcţii, compilatorul C++ va asocia automat orice eroare nespecificată în listă cu funcţia *unexpected.*

EXCEPŢII PE MAI MULTE NIVELURI

Când programele generează şi preiau excepţii, apar situaţii când o rutină de tratare realizează operaţii care necesită, la rândul lor, tratarea erorilor. Să considerăm, de exemplu, următorul fragment de program. Instrucţiunile activează tratarea erorilor folosind instrucţiunea *try*. Apoi, instrucţiunile apelează funcţia *some_operation*. Dacă operaţia lansează excepţia *out_of_memory*, rutina de tratare va prelua excepţia şi va derula instrucţiuni de tratare a unor excepţii suplimentare:

```
try {
     some_operation();
}
catch (out_of_memory) {
     // statements

  try {
       some_other_operation();
  }

  catch (cant_write_to_file) {
       // statements
  }
}
```

În funcţie de complexitatea programului, pot exista mai multe niveluri de excepţii incluse unele în altele.

REPETAREA UNEI OPERAŢII

Pentru simplitate, majoritatea programelor precedente se încheie la întâlnirea unei excepţii. În multe situaţii, însă, se doreşte rezolvarea cauzei erorii pentru a putea repeta operaţia ce a lansat eroarea. La începutul acestui capitol, programul ARRAYEXEC.CPP descoperea valori eronate de indici transmise funcţiilor *fill_array* şi *show_array*. Următorul program, TRYAGAIN.CPP, preia astfel de indici eronaţi şi decrementează continuu valoarea acestora, până când este găsită o valoare acceptabilă de index:

```
#include <iostream.h>
#include <stdlib.h>

class Array {
  public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);
    class range { };
  private:
    int *buffer;
    int size;
};

Array::Array(int size)
 {
   buffer = new int[size];
   Array::size = size;
 }

void Array::fill_array(int num_values)
 {
   if (num_values > size)
     throw range();

   for (int i = 0; i < num_values; i++)
     buffer[i] = i;
 }

void Array::show_array(int num_values)
 {
   if (num_values > size)
     throw range();

   for (int i = 0; i < num_values; i++)
     cout << buffer[i] << endl;
 }

void main(void)
```

```
{
  Array vector(200);
  int index = 500;

  fill:
  try {
    vector.fill_array(index);
  }
  catch (Array::range) {
    index -= 1;
    goto fill;
  }

  index = 500;

  show:
  try {
    vector.show_array(index);
  }
  catch (Array::range) {
    index -= 1;
    goto show;
  }
}
```

De fiecare dată când o excepţie este preluată pentru tratare, programul scade o unitate din indicele incorect, apoi foloseşte instrucţiunea *goto* pentru a se conecta la o locaţie ce precede instrucţiunea *try*, în scopul repetării operaţiei. Programul trebuie să transfere controlul înainte de instrucţiunea *try* pentru a permite tratarea excepţiilor pentru următorul apel de funcţie. Exersaţi cu acest program şi apelaţi funcţia din interiorul rutinei de tratare, cum se arată mai jos:

```
try {
  vector.fill_array(index);
}
catch (Array::range) {
  index -= 1;
  vector.fill_array(index);
}
```

Deoarece tratarea erorilor nu a fost reluată, programul va apela rutina prestabilită de tratare, după care se va încheia.

TRATAREA EXCEPŢIILOR NEPRELUATE

Aşa cum aţi învăţat, când o excepţie este lansată, dar omisă, se va executa o rutină de tratare prestabilită, care va încheia execuţia programului. Numele acestei rutine implicite este *terminate*. Următorul program, USE_TERM.CPP,

apelează pur şi simplu funcţia *terminate* pentru a verifica existenţa acesteia. Observaţi că programul include fişierul antet EXCEPT.H. În funcţie de compilatorul pe care îl aveţi la dispoziţie, numele fişierului antet ce conţine prototipul funcţiei *terminate* poate fi uşor diferit:

```
#include <iostream.h>
#include <except.h>

void main(void)
 {
   terminate();
 }
```

În mod prestabilit, funcţia *terminate* afişează un mesaj şi apoi încheie execuţia. În alte situaţii s-ar putea crea o funcţie proprie care să înlocuiască funcţia *terminate*. Acest lucru se poate realiza cu ajutorul funcţiei *set_terminate*. Următorul program, HASTA.CPP, foloseşte funcţia *set_terminate* pentru a apela funcţia *my_termination* în cazul unei excepţii generate, dar omise:

```
#include <iostream.h>
#include <except.h>
#include <stdlib.h>

class my_error { };
class your_error { };

void my_termination(void)
 {
   cerr << "Hasta la vista, baby!" << endl;
   exit(1);
 }

void some_operation(void)
 {
   throw your_error();
 }

void main(void)
 {
   set_terminate(my_termination);
   try {
     some_operation();
   }
   catch (my_error) {
     cerr << "Caught my error !" << endl;
   }
 }
```

În programul anterior, deoarece excepţia *your_error* este lansată, dar omisă, este apelată funcţia *terminate*, care este substituită de funcţia *my_termination*. Aceasta afişează un mesaj şi încheie execuţia programului prin funcţia *exit*.

Aşa cum aţi învăţat, dacă se specifică excepţiile ce pot fi lansate de o funcţie, dar această funcţie va genera ulterior o excepţie nespecificată, programul va apela o rutină prestabilită de tratare denumită *unexpected*. Următorul program, UNEXPECT.CPP, invocă funcţia *unexpected*, pentru a verifica existenţa acesteia:

```
#include <iostream.h>
#include <except.h>

void main(void)
 {
   unexpected();
 }
```

Există situaţii la generarea unei excepţii nespecificate când se doreşte apelarea altei funcţii, în locul funcţiei *unexpected*. Această substituţie se poate face cu ajutorul funcţiei *set_unexpected*. Programul următor, SET_UNEX.CPP, foloseşte funcţia *set_unexpected* pentru a instala o rutină proprie de tratare, care se execută când o funcţie lansează o excepţie nespecificată:

```
#include <iostream.h>
#include <except.h>
#include <stdlib.h>

class my_error { };
class your_error { };

void my_unexpected_handler(void)
 {
   cerr << "Shame on someone for throwing an unlisted exception!" <<
           endl;
   exit(1);
 }

void some_operation(void) throw (my_error)
 {
   throw your_error();
 }

void main(void)
 {
   set_unexpected(my_unexpected_handler);
   try {
     some_operation();
   }
   catch (my_error) {
     cerr << "Caught my error !" << endl;
   }
 }
```


REZUMAT

Tratarea excepţiilor permite programelor să preia şi să trateze o gamă largă de erori definite de program. Dacă compilarea programelor prezentate în acest capitol nu reuşeşte, încercaţi vă procuraţi o versiune mai nouă a compilatorului C++. În Capitolul 15 veţi învăţa cum să controlaţi şi să gestionaţi zona de memorie alocabilă (heap). Înainte de a trece la Capitolul 15, verificaţi dacă aţi învăţat următoarele:

- ✓ În cel mai simplu sens, o excepţie este o eroare definită de program. O rutină de tratare a excepţiei (handler) este o secvenţă de program scrisă pentru a răspunde unei astfel de erori.
- ✓ În C++, tratarea erorilor este realizată de instrucţiunile *try*, *throw* şi *catch*.
- ✓ Instrucţiunea *try* se foloseşte într-un program pentru a valida tratarea excepţiilor.
- ✓ Pentru a determina apariţia unei excepţii se foloseşte instrucţiunea *catch*.
- ✓ Pentru a determina apariţia unei anumite excepţii se foloseşte instrucţiunea *catch*.
- ✓ Pentru generarea unei excepţii, se foloseşte instrucţiunea *throw*. Cu alte cuvinte, când apare o eroare, programele lansează o excepţie ce va fi ulterior preluată de rutina de tratare corespunzătoare.
- ✓ Fiecare excepţie definită de program are un nume unic. De obicei, se foloseşte câte o clasă pentru fiecare excepţie. Prin utilizarea unei clase, programele pot atribui valori variabilelor membru ale clasei, pentru a furniza mai multe informaţii rutinei de tratare a excepţiei respective.
- ✓ Dacă o excepţie este lansată, dar omisă, se va executa o rutină de tratare prestabilită, denumită *terminate*. În majoritatea situaţiilor, funcţia *terminate* afişează un mesaj de eroare şi opreşte execuţia programului. Programele îşi pot defini propriile rutine de încheiere folosind funcţia *set_terminate*. Fişierul antet EXCEPT.H conţine prototipul funcţiei *set_terminate*.
- ✓ Aşa cum programele folosesc prototipuri de funcţii pentru specificarea tipului valorii unei funcţii şi a parametrilor ei, la fel poate fi specificată o listă de excepţii pe care le poate lansa o funcţie. Dacă o funcţie generează o excepţie ce nu este conţinută în listă, C++ va apela o rutină specială de tratare, denumită *unexpected*. Folosind funcţia *set_unexpected*, programele îşi pot defini propriile rutine de tratare a excepţiilor nespecificate. Fişierul antet EXCEPT.H conţine prototipul funcţiei *set_unexpected*, în funcţie de compilatorul folosit.

CAPITOLUL 15

APROFUNDAREA GESTIUNII ZONEI DE MEMORIE ALOCABILĂ

În Capitolul 7 aţi învăţat cum se alocă şi se eliberează dinamic memoria în C++, folosind operatorii *new* şi *delete*. Aţi învăţat că atunci când realizaţi alocarea dinamică a memoriei, operatorul *new* obţine memorie dintr-o zonă denumită *heap* (zonă de memorie alocabilă). Zona alocabilă este, aşadar, un ansamblu de locaţii de memorie aflat la dispoziţia programului. Pe măsură ce creşte complexitatea programelor ce realizează alocarea dinamică a memoriei, apar situaţii când trebuie să examinaţi diferite proprietăţi ale zonei alocabile, pentru a putea depista erorile dificile din program. Dacă lucraţi într-un mediu DOS, programele pot folosi funcţiile prezentate în acest capitol pentru examinarea zonei alocabile. Dacă lucraţi în alte medii, reţineţi conceptele prezentate în acest capitol, deoarece există funcţii similare şi în alte medii de lucru.

Când veţi termina acest capitol, veţi cunoaşte:

- Ce este zona alocabilă şi care este dimensiunea ei
- Cum se gestionează regiunile de memorie alocate şi libere
- Cum se pot detecta erorile de alocare a memoriei
- Cum se poate detecta alterarea zonei alocabile cauzată de eori de programare

Observaţie: *Compilatorul pe care îl folosiţi poate utiliza nume de funcţii diferite de cele din acest capitol, care se referă îndeosebi la compilatorul Borland C++.*

EXAMINAREA ZONEI DE MEMORIE ALOCABILĂ

Zona alocabilă este o regiune de memorie din care programele pot aloca dinamic porţiuni de memorie, cum se arată în Figura 15.1.

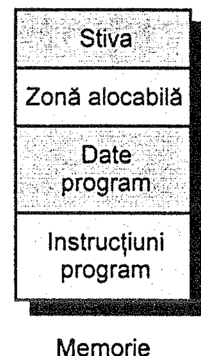


Figura 15.1 *Zona alocabilă (heap) este o porţiune a memoriei din care programele pot aloca memorie în mod dinamic.*

Presupunând că un program alocă trei regiuni în memorie cu dimensiunile de 100, 200 şi respectiv 300 de octeţi, zona alocabilă va arăta ca în Figura 15.2.

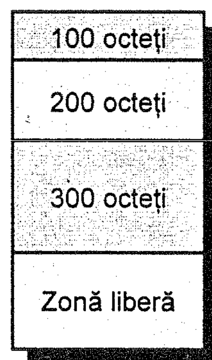


Figura 15.2 Alocarea memoriei în zona alocabilă .

Dacă ulterior programul va elibera 200 de octeţi, zona alocabilă va arăta ca în Figura 15.3.

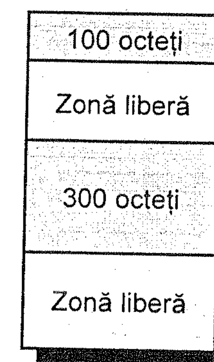


Figura 15.3. Eliberarea memoriei în zona alocabilă .

Pentru a permite alocarea dinamică a memoriei, sistemul de operare sau limbajul de programare furnizează funcţii de administrare a memoriei, a căror activitate este invizibilă operatorului uman. Pentru a aloca şi a elibera memorie din zona alocabilă , aceste funcţii urmăresc fiecare regiune care a fost alocată, precum şi pe cele disponibile. În cazul sistemului de operare DOS, majoritatea compilatoarelor creează o listă cu legături bazată pe structura *heapinfo* arătată în continuare:

```
struct heapinfo {
  void *ptr;
  unsigned int size;
  int in_use;
};
```

Lista cu legături conţine un nod *heapinfo* pentru fiecare regiune de memorie liberă sau folosită. Pointerul *ptr* indică următorul nod din lista înlănţuită. Componenta *size* specifică dimensiunea (în octeţi) a regiunii de memorie asociată nodului curent. Componenta *in_use* primeşte valoarea 1 dacă regiunea de memorie este alocată şi 0 dacă este disponibilă. Dată fiind alocarea anterioară de 100, 200 şi 300 de octeţi în zona alocabilă , lista cu legături va arăta ca în Figura 15.4.

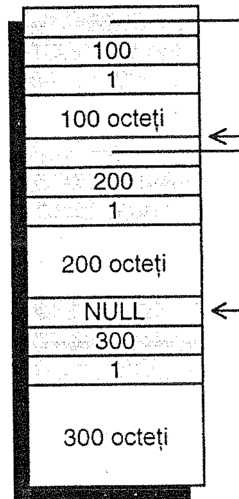


Figura 15.4 *Spațiul zonei alocabile este gestionat folosind o listă cu legături.*

Dacă programul ar urma să elibereze regiunea de 200 de octeți, valoarea componentei *in_use* a celui de al doilea nod se va schimba din 1 (folosit) la 0 (disponibil).

Pentru a vă ajuta să înțelegeți mai bine lista cu legături a zonei alocabile, următorul program, HEAPLIST.CPP, alocă memorie și apoi urmărește lista cu legături a zonei alocabile folosind funcția *heapwalk*. Programul eliberează apoi regiunile alocate și afișează noua listă cu legături:

```
#include <iostream.h>
#include <alloc.h>

void main(void)
 {
   char *bufferl, *buffer2, *buffer3, *buffer4;

   struct heapinfo node = {NULL, 0, 0};

   bufferl = new char[1000];
   buffer2 = new char[2000];
   buffer3 = new char[3000];
   buffer4 = new char[4000];

   while (heapwalk(&node) == _HEAPOK)
     {
       cout << "Size " << node.size << " bytes State ";
```



```
         if (node.in_use)
           cout << "In use" << endl;
         else
           cout << "Free" << endl;
       }

   node.ptr = NULL;  // Start at the beginning at the list

   delete buffer2;
   delete buffer4;

   cout << endl << "After memory release" << endl;

   while (heapwalk(&node) == _HEAPOK)
     {
       cout << "Size " << node.size << " bytes State ";

       if (node.in_use)
         cout << "In use" << endl;
       else
         cout << "Free" << endl;
     }
 }
```

La compilarea și execuția acesui program, pe ecran se vor afișa următoarele rezultate, care s-ar putea să vă deruteze:

```
C:\> HEAPLIST  <ENTER>
Size 516 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 14 bytes State In use
Size 16 bytes State In use
Size 8 bytes State In use
Size 1004 bytes State In use
Size 2004 bytes State In use
Size 3004 bytes State In use
Size 4004 bytes State In use

After memory release
Size 516 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
```

```
Size 520 bytes State In use
Size 14 bytes State In use
Size 16 bytes State In use
Size 8 bytes State In use
Size 1004 bytes State In use
Size 2004 bytes State Free
Size 3004 bytes State In use
```

Aşa cum se poate vedea, zona alocabilă conţine intrări şi pentru regiuni de memorie diferite de bufferele alocate de program. În funcţie de compilatorul folosit, motivul acestor alocări ar putea să fie diferit. De exemplu, alocarea ar putea fi efectuată automat pentru buffer-ele fluxurilor I/O. De asemenea, să observăm că în acest exemplu, nodul asociat fiecărei regiuni necesită 4 octeţi pentru regiuni de memorie *near* (care folosesc pointeri pe 16 biţi) şi 8 octeţi pentru regiuni de memorie *far* (care folosesc pointeri pe 32 biţi). Dacă folosiţi alt sistem de operare decât MS-DOS, compilaţi acest program folosind modelele de memorie *small* şi *large*. În cazul modelului de memorie *small*, nodul va folosi pointeri pe 16 biţi, iar în cazul modelului de memorie *large*, pe 32 de biţi.

CHEIA SUCCESULUI

EVIDENŢA LISTEI CU LEGĂTURI A ZONEI ALOCABILE

Zona alocabilă este o parte a memoriei pe care programele o alocă dinamic. Pe măsură ce programele alocă şi eliberează memorie, zona alocabilă ar putea conţine diferite porţiuni, unele în folosinţă iar altele libere. Pentru a controla regiunile zonei alocabile, majoritatea limbajelor de programare folosesc o listă cu legături ale cărei noduri corespund regiunilor zonei. Pentru a putea detecta erorile ce apar la programele care folosesc alocarea dinamică a memoriei, bibliotecile de subprograme furnizează un set de funcţii ce pot fi folosite la parcurgerea listei înlănţuite a zonei alocabile. Prin examinarea individuală a conţinutului zonei alocabile, se pot detecta multe erori de programare.

TESTAREA VALIDITĂŢII UNEI INTRĂRI DIN ZONA ALOCABILĂ

La alocarea memoriei în zona alocabilă , pot apărea situaţii când, datorită unei erori de programare, zona alocabilă este deteriorată prin suprascrierea unor date ale programului. Să presupunem, de exemplu, că un program alocă un buffer de 50 de octeţi şi alt buffer de 100 de octeţi, cum se arată în Figura 15.5.

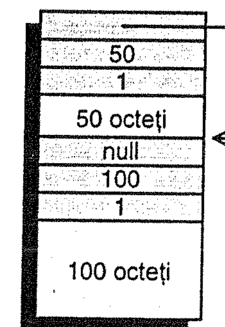


Figura 15.5 *Alocarea a două regiuni de memorie în zona alocabilă .*

Să presupunem, apoi, că programul foloseşte un ciclu *for* pentru a atribui valoarea 0 celor 50 de elemente ale primului buffer;

```
for (i = 0; i <= 50; i++)
   fifty[i] = 0;
```

Deoarece ciclul se repetă până când variabila *i* este mai mică sau egală cu 50, valoarea 0 va fi atribuită şi unei locaţii situată în afara bufferului. După cum se vede în Figura 15.6, această atribuire suprascrie un octet din componenta pointer a nodului asociat celei de a doua intrări în zona alocabilă .

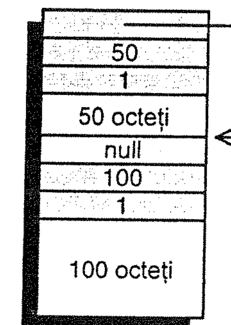


Figura 15.6 *Suprascrierea unui octet în pointerul altui nod datorată unei erori de programare similare.*

Următorul program, OVERWRIT.CPP, ilustrează o eroare de programare:

```
#include <iostream.h>
#include <string.h>
void main(void)
```

```
{
  char *fifty = new char[50];
  char *string = new char[100];

  strcpy(string, "Success with C++");

  for (int i = 0; i <= 60; i++)
    fifty[i] = 0;

  cout << string << endl;
}
```

Se observă că programul alocă două buffere de memorie şi atribuie celui de al doilea buffer un şir de caractere. Apoi, programul foloseşte un ciclu *for* pentru a atribui valoarea 0 fiecărui octet din primul buffer. După cum se vede în Figura 15.7, programul suprascrie o porţiune a celui de al doilea buffer:

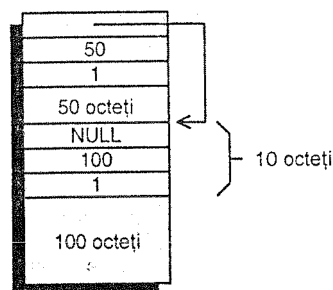


Figura 15.7 Suprascrierea unei regiuni a zonei alocabile.

Deoarece programul anterior este scurt, cauza erorii se poate detecta cu uşurinţă, folosind tehnicile de depanare obişnuite. Pe măsură, însă, ce programele devin mai complicate, este nevoie de a examina periodic intrările zonei alocabile, în scopul determinării locului apariţiei erorilor. Următoarele secţiuni prezintă câteva tehnici de programare folosite pentru a testa starea zonei alocabile.

CHEIA SUCCESULUI

SĂ ÎNŢELEGEM STAREA ZONEI ALOCABILE

Pentru a controla memoria din zona alocabilă, majoritatea limbajelor de programare folosesc o listă cu legături. Din nefericire, dacă un program conţine erori, el poate să suprascrie octeţi aparţinând nodurilor listei. În astfel de situaţii, zona alocabilă este deteriorată. Pentru a determina starea zonei alocabile, multe din bibliotecile de subprograme oferă funcţii ce ajută la examinarea tuturor intrărilor zonei alocabile, sau a anumitor intrări. În acest mod, programul poate să verifice dacă anumite intrări ale zonei sunt incorecte.

O VERIFICARE RAPIDĂ A ZONEI ALOCABILE

În funcţie de erorile care trebuie detectate, se poate dori, la un moment dat, verificarea validităţii intrărilor zonei alocabile. În astfel de situaţii, programele pot folosi funcţia de bibliotecă *heapcheck*. Această funcţie returnează una din valorile indicate în Tabela 15.1.

Valoare returnată	Explicaţia
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările în zonă sunt valide
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate

***Tabela 15.1** Valorile de stare ale zonei alocabile returnate de funcţia* heapcheck.

Următorul program, HEAPCHK.CPP, foloseşte funcţia *heapcheck* pentru a determina dacă zona alocabilă a fost deteriorată:

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>

void main(void)
 {
   char *fifty = new char[50];
   char *string = new char[100];

   strcpy(string, "Success with C++");

   switch (heapcheck()) {
    case _HEAPEMPTY:    cout << "No heap" << endl;
                        break;
    case _HEAPOK:       cout << "Heap is valid" << endl;
                        break;
    case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                        break;
   };

   for (int i = 0; i <= 60; i++)
     fifty[i] = 0;

   switch (heapcheck()) {
    case _HEAPEMPTY:    cout << "No heap" << endl;
                        break;
    case _HEAPOK:       cout << "Heap is valid" << endl;
                        break;
    case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                        break;
```

```
  };

  cout << string << endl;
}
```

După cum se observă, programul foloseşte funcţia *heapcheck* înainte şi după deteriorarea zonei alocabile de către ciclul *for*. La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> HEAPCHK  <ENTER>
Heap is valid
Heap is corrupted
```

Folosind funcţia *heapcheck* în acest mod se poate detecta locul exact din program unde zona alocabilă se alterează.

VALIDAREA UNEI ANUMITE INTRĂRI DIN ZONA ALOCABILĂ

La depanarea intrărilor deteriorate ale zonei alocabile, uneori este necesară examinarea de către program a fiecărui pointer al nodurilor listei. Pentru a determina validitatea intrărilor în zonă, programele pot folosi funcţia *heapchecknode*. Funcţia returnează una din valorile prezentate în Tabela 15.2.

Valoarea returnată	Explicaţia
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările în zonă sunt valide
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate
_BADNODE	Nodul curent este deteriorat
_FREENTRY	Blocul este liber
_USEDENTRY	Blocul este valid şi este folosit

Tabela 15.2 *Valorile de stare ale zonei alocabile returnate de funcţia* _heapchecknode.

Următorul program, NODECHK.CPP, foloseşte funcţia *heapchecknode* pentru a examina pointerii programului:

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>

void main(void)
 {
   char *fifty = new char[50];
   char *string = new char[100];
```



```
  strcpy(string, "Success with C++");

  for (int i = 0; i <= 60; i++)
    fifty[i] = 0;

  switch (heapchecknode(fifty)) {
   case _HEAPEMPTY:    cout << "No heap" << endl;
                       break;
   case _HEAPOK:       cout << "Heap is valid" << endl;
                       break;
   case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                       break;
   case _BADNODE:      cout << "Node is corrupted" << endl;
                       break;
   case _FREEENTRY:    cout << "Entry is not in use" << endl;
                       break;
   case _USEDENTRY:    cout << "Entry is valid and in use" << endl;
                       break;
  };

  switch (heapchecknode(string)) {
   case _HEAPEMPTY:    cout << "No heap" << endl;
                       break;
   case _HEAPOK:       cout << "Heap is valid" << endl;
                       break;
   case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                       break;
   case _BADNODE:      cout << "Node is corrupted" << endl;
                       break;
   case _FREEENTRY:    cout << "Entry is not in use" << endl;
                       break;
   case _USEDENTRY:    cout << "Entry is valid and in use" << endl;
                       break;
  };

  cout << string << endl;
}
```

Se observă că programul foloseşte funcţia *heapchecknode* pentru fiecare variabilă pointer. La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> NODECHK  <ENTER>
Entry is valid and in use
Heap is corrupted
```

TESTAREA SPAŢIULUI DE MEMORIE DISPONIBIL DIN ZONA ALOCABILĂ

La alocarea şi eliberarea, în timp, a memoriei, spaţiul disponibil poate deveni fragmentat, zona alocabilă fiind formată din spaţii libere şi spaţii alocate dispuse alternativ, cum se arată în Figura 15.8.

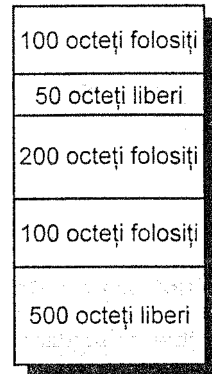


Figura 15.8 Spațiul liber fragmentat din memoria alocabilă.

Dacă un program suprascrie o porțiune de memorie disponibilă din zona alocabilă , efectul erorii poate să apară mai târziu, când programul va încerca să aloce zona de memorie disponibilă. Pentru a detecta astfel de erori, se pot folosi funcțiile *heapfillfree* și *heapcheckfree*. Funcția *heapfillfree* atribuie o valoare specifică tuturor locațiilor spațiului liber. De exemplu, programul ar putea atribui valoarea -1 fiecărei regiuni de memorie disponibilă. Ulterior, funcția *heapcheckfree* examinează spațiul liber al zonei alocabile pentru a se asigura că fiecare locație conține valoarea specificată. Dacă programul suprascrie o locație din memoria disponibilă, funcția *heapcheckfree* va detecta schimbarea, permițând programului să detecteze eroarea. Funcția *heapcheckfree* returnează una din valorile prezentate în Tabela 15.3.

Valoarea returnată	Explicația
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările din zonă sunt corecte
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate
_BADVALUE	O altă valoare a fost întâlnită într-o locație liberă

Tabela 15.3 Valorile de stare ale zonei alocabile returnate de funcția heapcheckfree.

Următorul program, CHKFREE.CPP, folosește funcția *heapfillfree* pentru a atribui valoarea -1 spațiului liber de memorie. Programul folosește apoi un ciclu *for* eronat pentru a suprascrie un buffer (din spațiul liber). La terminarea ciclului *for* programul detectează eroarea folosind funcția *heapcheckfree*:

```
#include <iostream.h>
#include <alloc.h>
```



```
void main(void)
{
  char *start, *middle, *end;
  int state;

  start = new char[50];
  middle = new char[50];
  end = new char[50];

  delete middle;     // Create free space in the middle

  state = heapfillfree(-1);

  if (state == _HEAPOK)
    cout << "Heap is ok" << endl;
  else if (state == _HEAPCORRUPT)
    cout << "Heap is corrupt" << endl;

  for (int i = 0; i <= 60; i++)
    start[i] = i;

  state = heapcheckfree('A');

  if (state == _HEAPOK)
    cout << "Heap is ok" << endl;
  else if (state == _HEAPCORRUPT)
    cout << "Heap is corrupt" << endl;
  else if (state == _BADVALUE)
    cout << "Value has been changed in free space" << endl;
}
```

După cum se observă, programul alocă trei buffere de memorie, apoi eliberează bufferul din mijloc pentru a crea spațiu liber. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> CHKFREE  <ENTER>
Heap is ok
Value has been changed in free space
```

COMPACTAREA SPAȚIULUI DE MEMORIE DISPONIBIL

Cu timpul, spațiul de memorie liber și cel alocat pot deveni alocabile. Să presupunem, de exemplu, că zona alocabilă apare ca în Figura 15.8. Dacă programul eliberează regiunea de 200 de octeți, funcțiile de administrare a zonei vor compacta cele două regiuni în una singură, cum se arată în Figura 15.9.

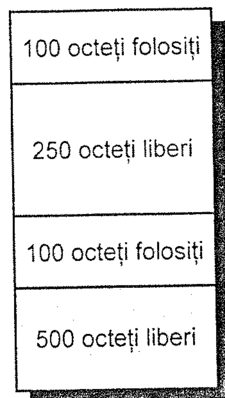


Figura 15.9 Compactarea spațiului disponibil din zona alocabilă .

Aşa cum se vede în Figura 15.9, zona alocabilă conţine 750 de octeţi de spaţiu disponibil. Deoarece spaţiul este fragmentat, cea mai mare porţiune de memorie pe care programul o poate aloca este de 500 de octeţi. Din păcate compilatorul C++ nu realizează *colectarea resturilor*, care ar duce, periodic, la unificarea tuturor spaţiilor libere, cum se arată în Figura 15.10.

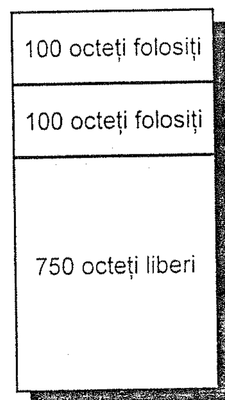


Figura 15.10 Colectarea resturilor este procesul de compactare a spațiului disponibil din zona alocabilă .

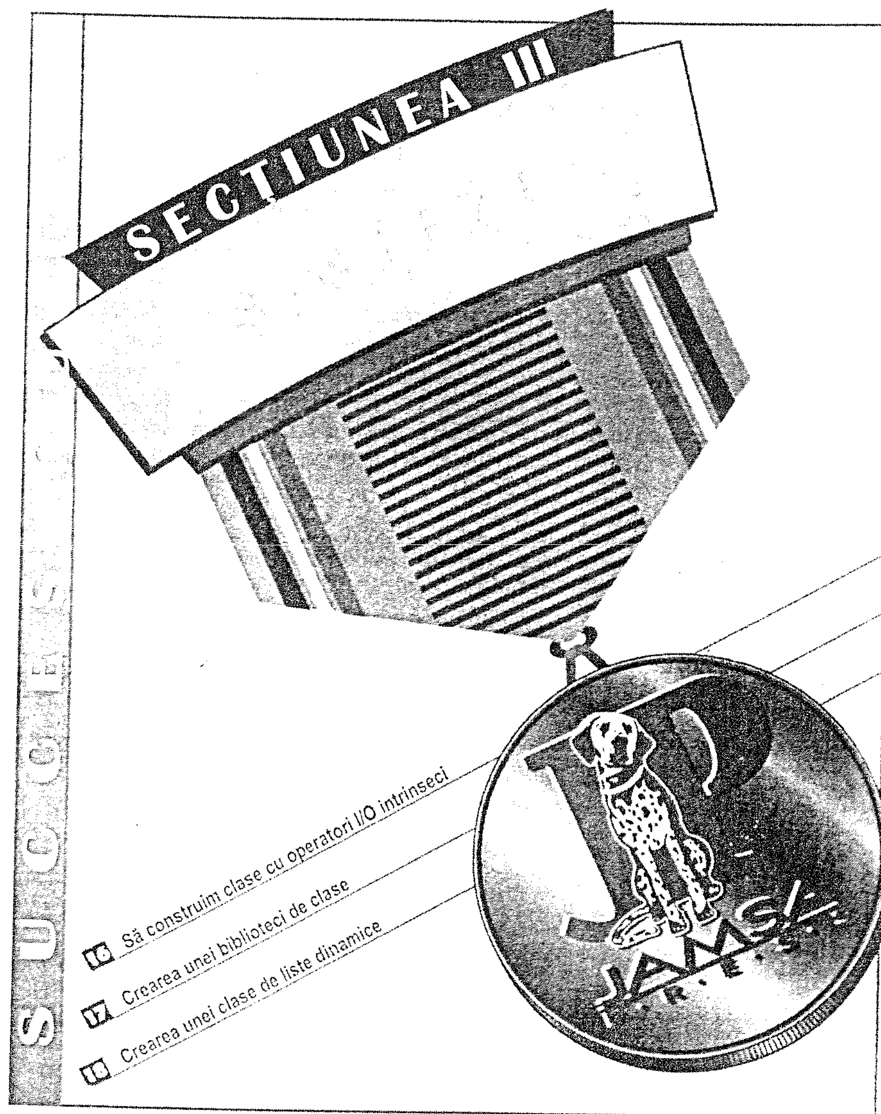
Deoarece programele conţin pointeri la anumite regiuni de memorie, rutinele de administrare a zonei alocabile nu pot deplasa regiunile de memorie pentru a

compacta spaţiul disponibil (rutinele de gestiune nu au cunoştinţă de variabilele cărora li s-au atribuit valorile pointerilor, iar dacă memoria este deplasată pointerii vor fi eronaţi). Dacă un program nu poate aloca spaţiu datorită fragmentării, atunci programele vor elibera şi realoca memorie succesiv, realizând astfel propria colectare de resturi.

REZUMAT

Când programele alocă şi eliberează memorie în mod dinamic, folosind operatorii *new* şi *delete*, pot apărea erori greu de detectat. Pentru a contribui la depanarea unor astfel de erori, majoritatea compilatoarelor sau sistemelor de operare furnizează un set de funcţii de bibliotecă ce permit testarea stării zonei alocabile. Folosiţi aceste funcţii ori de câte ori aveţi necazuri cu alocarea dinamică a memoriei. Înainte de a trece la Capitolul 16, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Zona alocabilă este o regiune de memorie pe care programele o alocă dinamic. În funcţie de sistemul de operare sau de modelul de memorie a compilatorului, dimensiunea acestei zone variază.
- ✓ Pentru gestiunea spaţiilor de memorie disponibile sau alocate, se foloseşte o listă cu legături. Pentru fiecare intrare, zona alocabilă controlează dimensiunea şi disponibilitatea acesteia (dacă zona este sau nu disponibilă).
- ✓ Pentru a detecta o eroare de alocare a memoriei, programele pot parcurge intrările listei zonei alocabile.
- ✓ Prin examinarea intrărilor zonei alocabile utilizând acest procedeu, programele pot detecta deteriorările locaţiilor de memorie cauzate de erori de programare.


SECȚIUNEA III
SUCCESE
16 Să construim clase cu operatori I/O intrinseci
17 Crearea unei biblioteci de clase
18 Crearea unei clase de liste dinamice
JAMSA PRESS

CAPITOLUL 16
SĂ CONSTRUIM CLASE CU OPERATORI I/O INTRINSECI

Multe din programele din această carte au folosit funcţii membru pentru afişarea variabilelor membru ale unei clase, cum au fost *show_class*, *show_book*, *show_employee*. Urmărind acest model, programele ar putea folosi funcţii membru ca *file_class* sau *file_employee* pentru a transcrie membrii unei clase într-un stream fişier. Când proiectaţi programe, acestea vor deveni mai uşor de înţeles prin scrierea unor clase cu operatori I/O intrinseci.

În cel mai simplu sens, o clasă cu operatori I/O intrinseci este o clasă ce suportă operatorii de inserţie (<<) şi extracţie (>>). Folosind aceşti operatori I/O, programele pot uşor realiza operaţii I/O cu tastatura, ecranul sau fişierele, aşa cum se folosesc stream-urile I/O *cin* şi *cout*. Acest capitol examinează în detaliu clasele cu operatori I/O intrinseci. În particular, veţi învăţa:

- Ce este şi cum se creează o clasă cu operatori I/O intrinseci
- Cerinţele legate de operatorii suprapuşi
- Cum suprapunerea operatorilor pentru crearea claselor cu operatori I/O intrinseci este similară creării unui manipulator
- Unde se declară funcţia apelată de operatorul supradefinit
- Ce tip are rezultatul şi primul parametru al funcţiei apelate de operatorul suprapus
- Cum se defineşte tipul unei clase cu operatori I/O intrinseci
- Despre clasele cu operatori I/O intrinseci şi operaţiile I/O pe fişiere
- Cum se folosesc operatorii de inserţie şi extracţie

INTRĂRI/IEŞIRI CU MEMBRII CLASEI ÎN STIL CLASIC

Majoritatea programelor C++ folosesc funcţii membru pentru a realiza operaţii I/O cu tastatura, ecranul sau fişierele. De exemplu, următorul program, TRAVEL.CPP, foloseşte clasa *travel*:

```
class travel {
  public:
    travel(char *destination, int distance);
    void display_trip(void);
    int file_trip(ofstream& file);
  private:
    char destination[64];
    int distance;
};
```

Aşa cum se observă, clasa foloseşte funcţiile membru *display_trip* şi *file_trip* pentru a efectua operaţii de ieşire cu ecranul şi pe fişiere. Următoarele instrucţiuni implementează TRAVEL.CPP:

```
#include <fstream.h>
#include <string.h>

class travel {
  public:
    travel(char *destination, int distance);
    void display_trip(void);
    int file_trip(ofstream& file);
  private:
    char destination[64];
    int distance;
};

travel::travel(char *destination, int distance)
 {
   strcpy(travel::destination, destination);
   travel::distance = distance;
 }

void travel::display_trip(void)
 {
   cout << "Destination: " << destination << " Distance: " <<
        distance << endl;
 }

int travel::file_trip(ofstream& out_file)
 {
   out_file << destination << endl;
   out_file << distance << endl;
   return(1);
 }

void main(void)
 {
   travel vacation("Maui", 2500);
```



```
   vacation.display_trip();

   ofstream trip_file("TRAVEL.DAT");

   vacation.file_trip(trip_file);
 }
```

Programul apelează cele două funcţii membru pentru a realiza operaţii I/O. La compilarea şi execuţia programului, pe ecran se va afişa:

```
C:\> TRAVEL  <ENTER>
Destination: Maui Distance: 2500
```

Este important de observat că majoritatea programelor C++ folosesc funcţii membru pentru a realiza operaţii I/O bazate pe clase. Problema care apare la folosirea acestor funcţii membru este aceea că, la citirea programului, trebuie înţelese funcţiile suplimentare.

O clasă cu operatori I/O intrinseci, pe de altă parte, foloseşte operatorul de inserţie pentru a realiza astfel de operaţii I/O:

```
cout << vacation;

output_file << business_trip;
```

Pentru a realiza operaţii I/O în acest mod, programele trebuie să suprapună operatorul de inserţie (<<) pentru fiecare tip de clasă în parte. De exemplu, următorul program, STREAM.CPP, foloseşte operatorul de inserţie pentru a afişa obiectul *travel* şi pentru a scrie informaţii despre călătorie într-un fişier:

```
#include <fstream.h>
#include <string.h>

class travel {
  public:
    travel(char *destination, int distance);
    friend ostream& operator <<(ostream& stream, travel object);
    friend ofstream& operator <<(ofstream& stream, travel object);
  private:
    char destination[64];
    int distance;
};

travel::travel(char *destination, int distance)
 {
   strcpy(travel::destination, destination);
   travel::distance = distance;
 }

ostream& operator <<(ostream& stream, travel object)
 {
```

```
  stream << "Destination: " << object.destination <<
       " Distance: " << object.distance << endl;

  return(stream);
}

ofstream& operator <<(ofstream& stream, travel object)
{
  stream << object.destination << endl;
  stream << object.distance << endl;

  return(stream);
}


void main(void)
{
  travel vacation("Maui", 2500);

  cout << vacation;

  ofstream trip_file("TRAVEL.DAT");

  trip_file << vacation;
}
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> STREAM  <ENTER>
Destination: Maui  Distance: 2500
```

Majoritatea operaţiilor realizate de acest program au fost discutate în capitolele precedente. Totuşi, dacă nu înţelegeţi programul, nu vă îngrijoraţi, deoarece vom discuta detaliat în acest capitol fiecare pas al programului. Deocamdată, însă, să înţelegem că o clasă cu operatori I/O intrinseci este o clasă ce foloseşte operatorii de inserţie şi extracţie pentru a realiza operaţii I/O bazate pe stream.

Observaţie: *La suprapunerea operatorului de inserţie sau extracţie pentru o anumită clasă, nu se va schimba destinaţia operatorului pentru alte clase. Astfel, programul precedent poate folosi operatorul de inserţie, de exemplu, pentru a scrie caractere pe ecran, în ciuda faptului că programul a suprapus acest operator pentru clasa* travel. *În acest mod, suprapunerea operatorului de inserţie pentru o anumită clasă nu diferă faţă de suprapunerea operatorului de adunare sau scădere, care va afecta numai anumite tipuri.*



SĂ ÎNŢELEGEM CLASELE CU OPERATORI I/O INTRINSECI

Majoritatea programelor C++ folosesc funcţii membru speciale pentru a realiza I/O operaţii cu tastatura, ecranul sau fişierele. Din păcate, astfel de funcţii introduc alte funcţii care trebuie înţelese de către cei care doresc să citească programul. O clasă cu operatori I/O intrinseci, pe de altă parte, furnizează funcţii ce acceptă operatorii de inserţie (<<) şi extracţie (>>). Prin crearea unor astfel de clase, intrările/ieşirile de clase ale programului se pot realiza într-un mod mai natural, prin operatorii de inserţie şi extracţie.

CREAREA UNEI CLASE CU OPERATORI I/O INTRINSECI

Pentru crearea unei astfel de clase, programele suprapun operatorii de inserţie (<<) şi extracţie (>>) pentru anumite stream-uril/O. De exemplu, să presupunem că un program foloseşte clasa *date* arătată în continuare:

```
class date {
   public:
       date(int month, int day, int year);
   private:
        int month;
        int day;
        int year;
};
```

Pentru a suprapune operatorul de inserţie în vederea unor operaţii de ieşire pe stream-uri cum este *cout*, trebuie suprapus operatorul << astfel:

```
class date {
  public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
  private:
    int month;
    int day;
    int year;
};
```

După cum se observă, suprapunerea operatorului de inserţie este similară creării unui manipulator, aşa cum s-a discutat în Capitolul 11. Când se suprapune un operator de inserţie sau extracţie, se defineşte pur şi simplu o funcţie ce se execută când operatorul este folosit cu un anumit tip de date. Tipul valorii returnate şi primul parametru ai funcţiei sunt întotdeauna de tip stream:

Tip valoare returnată
Primul parametru al funcției

```
ostream& operator <<(ostream& stream, date date_object)
 {
     // Statements
 }
```

Al doilea parametru al funcției specifică tipul de date pentru care se suprapune operatorul:

```
ostream& operator <<(ostream& stream, date date_object)
 {
     // Statements
 }
```

Al doilea parametru specifică tipul de date

Să observăm că funcția nu este bazată pe clasă, adică nu este o funcție membru a clasei. Totuși, în cadrul clasei, operatorul trebuie declarat *friend*, cum se arată mai jos:

```
class date {
  public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
  private:
    int month;
    int day;
    int year;
};
```

Specificarea operatorului ca friend

Prin declararea operatorului ca *friend*, el va avea acces la membrii de tip *private* ai clasei. Următorul program, DATE_STR.CPP folosește operații de ieșire pe stream pentru a afișa membrii clasei *date*:

```
#include <iostream.h>

class date {
  public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
  private:
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
 {
   date::month = month;
   date::day = day;
   date::year = year;
 }
```



```
ostream& operator <<(ostream& stream, date date object)
 {
   switch (date object.month) {
     case  1: stream << "January ";
              break;
     case  2: stream << "February ";
              break;
     case  3: stream << "March ";
              break;
     case  4: stream << "April ";
              break;
     case  5: stream << "May ";
              break;
     case  6: stream << "June ";
              break;
     case  7: stream << "July ";
              break;
     case  8: stream << "August ";
              break;
     case  9: stream << "September ";
              break;
     case 10: stream << "October ";
              break;
     case 11: stream << "November ";
              break;
     case 12: stream << "December ";
              break;
   };

   stream << date object.day << ", " << date object.year << endl;

   return(stream);
 }

void main(void)
 {
   date birthday(9, 30, 94);

   cout << birthday;
 }
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> DATE_STR  <ENTER>
September 30, 94
```



REGULI PENTRU CREAREA DE CLASE CU OPERATORI I/O INTRINSECI

Pentru a crea o clasă cu operatori I/O intrinseci, programul trebuie să suprapună operatorii de inserţie (<<) şi extracţie (>>) pentru o anumită clasă. Suprapunerea acestor operatori se face prin definirea unei funcţii ce este apelată de fiecare dată când operatorul este folosit cu un anumit tip de clasă. Când creaţi funcţia, reţineţi următoarele reguli:

- Declaraţi funcţia în afara clasei - funcţia operator nu este membru al clasei.
- Primul parametru al funcţiei, ca şi tipul rezultatului, trebuie să fie un stream.
- Al doilea parametru al funcţiei este tipul clasei.
- Dacă operatorul va avea acces la membrii de tip *private* ai clasei, trebuie să fie declarat *friend* în cadrul clasei.

OPERAŢII DE INTRARE PE STREAM

Paşii efectuaţi de un program pentru a realiza operaţii de intrare pe stream sunt aproape similari celor discutaţi anterior. Deosebirea este că trebuie suprapus operatorul de extracţie (>>). Următoarele instrucţiuni ilustrează modul în care programul suprapune operatorul de extracţie pentru clasa *date*:

```
istream& operator >>(istream& stream, date *date_object)
 {
   cout << "Please type in mm dd yy: " ;

   stream >> date_object->month;
   stream >> date_object->day;
   stream >> date_object->year;

   return(stream);
 }
```

După cum se observă, instrucţiunile respectă aceleaşi reguli discutate anterior. Următorul program, GET_DATE.CPP, foloseşte instrucţiuni de intrare pe stream pentru a introduce componentele unui obiect de tipul *date* - luna, ziua şi anul:

```
#include <iostream.h>

class date {
  public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
    friend istream& operator >>(istream& stream, date *date_object);
  private:
```



```
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
 {
   date::month = month;
   date::day = day;
   date::year = year;
 }

ostream& operator <<(ostream& stream, date date_object)
 {
   stream << date_object.month << '/' << date_object.day << '/' <<
     date_object.year << endl;

   return(stream);
 }

istream& operator >>(istream& stream, date *date_object)
 {
   cout << "Please type in mm dd yy: " ;

   stream >> date_object->month;
   stream >> date_object->day;
   stream >> date_object->year;

   return(stream);
 }

void main(void)
 {
   date birthday(9, 30, 94);

   cin >> &birthday;

   cout << birthday;
 }
```

La execuţia programului se va cere introducerea unei date. Programul foloseşte apoi operatorul de extracţie pentru a realiza operaţia de intrare.

OPERAŢII I/O PE STREAM FIŞIER

Clasele cu operatori I/O intrinseci nu se rezumă numai la stream-urile *cout* şi *cin*. De exemplu, următorul program, FILEDATE.CPP, foloseşte operaţii I/O pe stream fişier pentru a scrie date pe fişierul DATES.DAT. Aici programul suprapune operatorul de inserţie (<<) pentru obiectele clasei *ofstream*:

```
#include <fstream.h>

class date {
  public:
    date(int month, int day, int year);
    friend ofstream& operator <<(ofstream& stream,
      date date_object);
  private:
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
 {
   date::month = month;
   date::day = day;
   date::year = year;
 }

ofstream& operator <<(ofstream& stream, date date_object)
 {

   stream << date_object.month << ' ' << date_object.day << ' ' <<
     date_object.year << endl;

   return(stream);
 }

void main(void)
 {
   date birthday(9, 30, 94);
   date christmas(12, 25, 94);
   date new_years(1, 1, 95);

   ofstream datefile("DATES.DAT");

   datefile << birthday;
   datefile << christmas;
   datefile << new_years;
 }
```

La compilarea şi execuţia acestui program se vor scrie trei date pe fişierul DATES.DAT. Într-o manieră similară, următorul program READDATE.CPP suprapune operatorul de extracţie pentru a citi date dintr-un fişier până la întâlnirea sfârşitului de fişier. Programul suprapune aici operatorul de extracţie pentru obiecte de tipul *ifstream*:

```
#include <fstream.h>

class date {
```



```
  public:
    date(void) { };
    friend ifstream& operator >>(ifstream& stream,
      date *date_object);
    friend ostream& operator <<(ostream& stream, date date_object);
  private:
    int month;
    int day;
    int year;
};

ifstream& operator >>(ifstream& stream, date *date_object)
 {
   stream >> date_object->month >> date_object->day >>
     date_object->year;

   return(stream);
 }

ostream& operator <<(ostream& stream, date date_object)
 {
   stream << date_object.month << '/' << date_object.day << '/' <<
     date_object.year << endl;

   return(stream);
 }

void main(void)
 {
   ifstream datefile("DATES.DAT");
   date dateinfo;

   while (! datefile.eof())
     {
       datefile >> &dateinfo;
       if (! datefile.eof())
         cout << dateinfo;
     }
 }
```

După cum se observă, regulile de suprapunere a operatorilor de inserţie şi extracţie pentru operaţii I/O bazate pe fişiere nu sunt complicate.

ATENŢIE LA FIŞIERELE BINARE

Reţineţi că nu trebuie să folosiţi operatorii de inserţie şi extracţie pentru a scrie valori binare (de obicei, valori de tip real) pe un fişier. Aşa cum cunoaşteţi deja, operatorul de inserţie va converti astfel de valori în caractere ASCII, care pot cauza erori la introducerea valorilor. În schimb, trebuie folosite funcţiile membru *read* şi *write* pentru a realiza operaţii I/O. De exemplu, următorul program,

FLOATVAL.CPP, suprapune operatorul de inserţie pentru a scrie câteva structuri ce conţin valori reale în fişierul FLOATVAL.DAT. Pentru a afişa valorile, programul foloseşte funcţia membru *write*:

```
#include <fstream.h>

class values {
  public:
    values(float a, float b, float c);
    friend ofstream& operator <<(ofstream& stream,
      values *value_object);
  private:
    float a;
    float b;
    float c;
};

values::values(float a, float b, float c)
 {
  values::a = a;
  values::b = b;
  values::c = c;
 }

ofstream& operator <<(ofstream& stream, values *value_object)
 {
   stream.write((char *) value_object, sizeof(class values));

   return(stream);
 }

void main(void)
 {
   ofstream value_file("FLOATVAL.DAT");

   values a(1.1, 2.2, 3.3);
   values b(11.11, 22.22, 33.33);
   values c(111.111, 222.222, 333.333);

   value_file << &a;
   value_file << &b;
   value_file << &c;
 }
```

În mod similar, următorul program, FLOATIN.CPP, suprapune operatorul de extracţie pentru a citi valori reale din fişierul FLOATVAL.DAT într-o variabilă de tip structură. Programul va continua să citească şi să afişeze valorile membrilor structurii până la întâlnirea sfârşitului de fişier:

```
#include <fstream.h>

class values {
```



```
  public:
    values(void) {};
    friend ifstream& operator >>(ifstream& stream,
      values *value_object);
    friend ostream& operator <<(ostream& stream,
      values *value_object);
  private:
    float a;
    float b;
    float c;
};

ifstream& operator >>(ifstream& stream, values *value_object)
 {
   stream.read((char *) value_object, sizeof(class values));

   return(stream);
 }

ostream& operator <<(ostream& stream, values *value_object)
 {
   stream << value_object->a << ' ' << value_object->b << ' ' <<
     value_object->c << endl;

   return(stream);
 }

void main(void)
 {
   ifstream value_file("FLOATVAL.DAT");

   values a, b, c;

   value_file >> &a;
   cout << &a;

   value_file >> &b;
   cout << &b;

   value_file >> &c;
   cout << &c;
 }
```

La compilarea şi execuţia acestui program, pe ecran va apărea:

```
C:\> FLOATIN  <ENTER>
1.1 2.2 3.3
11.11 22.22 33.33
111.111 222.222 333.333
```

REZUMAT

Clasele cu operatori I/O intrinseci facilitează înţelegerea programelor. Aşa cum aţi văzut, crearea şi folosirea unor astfel de clase nu este dificilă. Înainte de a trece la următorul capitol, asiguraţi-vă că aţi învăţat următoarele:

- ✓ Pentru a crea o clasă cu operatori I/O intrinseci se suprapun operatorii de inserţie (<<) şi extracţie (>>).
- ✓ Suprapunerea operatorilor pentru crearea unei clase cu operatori I/O intrinseci nu diferă de suprapunerea operatorilor de adunare sau scădere; suprapunerea afectează un anumit tip de date.
- ✓ Operatorul suprapus trebuie declarat *friend* în cadrul clasei pentru a putea avea acces la membrii de tip *private* ai clasei.
- ✓ Suprapunerea operatorilor pentru crearea claselor cu operatori I/O intrinseci este similară creării unui manipulator - se defineşte o funcţie ce se execută când operatorul este folosit cu un anumit tip de date.
- ✓ Funcţia apelată de operatorul suprapus nu este o funcţie membru a clasei, ci este declarată în afara clasei.
- ✓ Valoarea returnată şi primul parametru al unei funcţii apelate de un operator suprapus trebuie să fie de tipul stream.
- ✓ Clasa pentru care se creează operatori I/O intrinseci este definită de al doilea parametru al funcţiei apelate de operatorul suprapus.
- ✓ La fel ca *cin* şi *cout*, se pot folosi şi stream-urifişier pentru operaţii I/O intrinseci claselor, prin simpla suprapunere a operatorilor pentru obiecte de tipul *ofstream*.
- ✓ Operatorii de inserţie şi extracţie nu trebuie folosiţi cu date binare. În locul lor se folosesc funcţiile membru *read* şi *write*.

CAPITOLUL 17
CREAREA UNEI BIBLIOTECI DE CLASE

În sensul cel mai simplu, o *bibliotecă* este o colecţie de funcţii folosite de programe pentru a realiza anumite sarcini. Biblioteca standard C/C++, de exemplu, conţine funcţii ce permit programelor să prelucreze şiruri de caractere, să lucreze cu fişiere, să aloce memorie ş.a.m.d. Pentru a folosi o bibliotecă, trebuie inclus, de obicei, un fişier antet la începutul programului pentru a specifica prototipurile de funcţii. Apoi, la editarea programului, se specifică numele bibliotecii, pentru a permite editorului de legături să localizeze funcţiile. În cazul unei biblioteci standard, editorul de legături caută automat funcţiile în anumite fişiere. Folosirea bibliotecilor elimină nevoia de a scrie şi a testa un volum considerabil de cod, deoarece funcţiile din bibliotecă sunt testate şi funcţionează corect.

O *bibliotecă de clase* este o bibliotecă ce conţine funcţiile membru ale unor anumite clase. Pe măsură ce construiţi definiţiile de clase, veţi constata şi o anumită clasă, necesară unui program, este adesea utilă şi altui program. Punând împreună toate funcţiile membru într-o bibliotecă de clase, refolosirea claselor în două sau mai multe programe devine foarte simplă. Mulţi proiectanţi de software livrează biblioteci de clase. De exemplu, dacă programaţi în C++ pentru mediul Windows, există biblioteci de clase ce vă permit să realizaţi operaţii cu ferestre. Există, de asemenea, biblioteci de clase pentru aplicaţii multimedia. În fiecare caz, bibliotecile conţin funcţii membru şi definiţii asociate unor clase C++.

Acest capitol examinează etapele ce trebuie efectuate pentru a crea propriile biblioteci de clase. În momentul încheierii acestui capitol, veţi cunoaşte:

- Ce este, cum se creează şi cum se foloseşte o bibliotecă de programe obiect
- Ce este o bibliotecă de clase, cum se creează şi cum se foloseşte.

CREAREA ŞI FOLOSIREA UNEI BIBLIOTECI DE OBIECTE

Înainte de a examina etapele necesare creării unei biblioteci de clase, este util de a înţelege modul de creare a unei *biblioteci de obiecte*. O bibliotecă de obiecte este un fişier ce conţine codul compilat al unui set de funcţii. La crearea programelor se pot folosi aceste funcţii precompilate pentru a economisi timp,

efort de programare şi testare. Să construim acum biblioteca de obiecte STRFUNC.LIB ce conţine funcţiile *string_copy* şi *string_length*. Pentru început, creăm fişierul STRFUNC.CPP ce conţine următoarele:

```
void string_copy(char *target, char *source)
 {
   while (*target++ = *source++)
     ;
 }

int string_length(char *string)
 {
   for (int length = 0; *string++; length++)
     ;

   return(length);
 }
```

Se compilează apoi acest fişier sursă pentru a obţine fişierul obiect STRFUNC.OBJ. Dacă folosiţi Borland C++, apelaţi la următoarea linie de comandă pentru a crea fişierul obiect:

```
C:\> BCC -c STRFUNC.CPP <ENTER>
```

Dacă folosiţi Microsoft Visual C++, atunci linia de comandă va fi:

```
C:\> CL -c STRFUNC.CPP <ENTER>
```

Apoi, folosind programe de bibliotecă furnizate de compilator, creaţi fişierul de bibliotecă STRFUNC.LIB. Pentru Borland C++, folosiţi următoarea comandă TLIB pentru a crea fişierul de bibliotecă:

```
C:\> TLIB STRFUNC.LIB +STRFUNC.OBJ <ENTER>
```

Dacă lucraţi cu Microsoft Visual C++, folosiţi comanda LIB:

```
C:\> LIB STRFUNC.LIB +STRFUNC.OBJ; <ENTER>
```

Observaţie: *La folosirea utilitarului Microsoft LIB, nu uitaţi să scrieţi simbolul ';' la sfârşitul liniei de comandă, ceea ce va determina folosirea unor parametri prestabiliţi. În caz contrar, se va cere introducerea unor nume de fişiere.*

Pentru facilita specificarea prototipurilor de funcţii, se poate crea un fişier antet ce conţine prototipurile funcţiilor de bibliotecă. Să creăm, deci, fişierul antet STRFUNC.H ce conţine următoarele prototipuri:

```
void string_copy(char *target, char *source);

int string_length(char *string);
```

Următorul program, STRDEMO.CPP, foloseşte funcţiile de bibliotecă *string_length* şi *string_copy*:


```
#include <iostream.h>
#include "strfunc.h"          // string library header

void main(void)
 {
   char source[64] = "Success with C++";
   char target[64];
   char chapter[64];

   cout << "The string " << source << " has " <<
           string_length(source) << " characters" << endl;

   string_copy(chapter, "Chapter 17");

   cout << "This is " << chapter << endl;

   string_copy(target, source);

   cout << "I am reading " << target << endl;
 }
```

Dacă compilaţi (sau legaţi) acest program, trebuie specificat numele bibliotecii STRFUNC.LIB. Pentru compilatorul Borland C++, se poate compila şi lega programul folosind următoarea comandă:

```
C:\> BCC STRDEMO.CPP STRFUNC.LIB <ENTER>
```

Pentru compilatorul Microsoft Visual C++, compilarea şi legarea se realizează astfel:

```
C:\> CL STRDEMO.CPP STRFUNC.LIB <ENTER>
```

La execuţia programului, pe ecran va apărea:

```
C:\> STRDEMO <ENTER>
The string Success with C++ has 16 characters
This is Chapter 17
I am reading Success with C++
```



CREAREA UNEI BIBLIOTECI DE OBIECTE

O bibliotecă de obiecte este un fişier ce conţine codul compilat al unui set de funcţii. La crearea programelor, aceste funcţii precompilate pot fi folosite pentru a economisi timp şi efort de programare şi testare. Pentru a crea o bibliotecă de obiecte proprie, realizaţi paşii următori:

1. Creaţi un fişier sursă ce conţine funcţii asociate.
2. Compilaţi codul sursă pentru a produce un fişier obiect.

3. Folosiţi un program utilitar pentru a crea fişierul bibliotecă.
4. Creaţi un fişier antet ce conţine prototipurile fiecărei funcţii de bibliotecă.

Pentru a folosi funcţiile de bibliotecă în programe, urmaţi paşii de mai jos:

1. Includeţi fişierul antet corespunzător la începutul programului.
2. Folosiţi în programul sursă una sau mai multe funcţii de bibliotecă.
3. Legaţi programul cu fişierul bibliotecă.

CREAREA UNEI BIBLIOTECI DE CLASE

Aşa cum aţi învăţat, o bibliotecă de cod obiect conţine una sau mai multe funcţii ce pot fi legate la program. În mod asemănător, o bibliotecă de clase conţine codul obiect al uneia sau mai multor funcţii. Într-o bibliotecă de clase, însă, funcţiile corespund unei clase sau unei mulţimi de clase asociate. Aşa cum veţi vedea, algoritmul ce trebuie urmat pentru a crea o bibliotecă de clase nu diferă de cel discutat anterior pentru crearea unei biblioteci de cod obiect:

1. Plasaţi declaraţia clasei şi funcţiile membru în fişierul sursă.
2. Compilaţi fişierul sursă pentru a crea un fişier obiect.
3. Folosiţi un program utilitar pentru a crea fişierul bibliotecă.
4. Creaţi un fişier antet ce conţine declaraţia clasei.

Pentru a folosi în program biblioteca de clase, realizaţi următoarele:

1. Includeţi fişierul antet corespunzător la începutul programului.
2. Utilizaţi unul sau mai multe obiecte în program.
3. Compilaţi fişierul sursă al programului şi legaţi-l cu fişierul de bibliotecă al claselor.

După cum cunoaşteţi, multe programe folosesc interfeţe pe bază de meniuri. În acest scop s-ar putea crea o bibliotecă cu o clasă *menu* ce ar putea fi folosită de multe alte programe. Următoarea declaraţie de exemplu, creează o clasă *menu*:

```
struct option {              // each menu option has option
  char menu_text[64];        // text and a related keystroke
  char keystroke;
};

class menu {
  public:
    menu(char *title, option *menu_options, int foreground,
      int background);
    void show_menu(void);
    int get_option(void);
  private:
```



```
    char title[64];            // menu title
    option options[7];         // array of menu option strings
    int number_of_options;     // number of menu options
    int foreground;            // menu foreground color
    int background;            // menu background color
    int row;                   // row to start menu display
    int column;                // column to start menu display
    void set_colors(int foreground, int background);
    void set_row_and_column(int row, int column);
};
```

Clasa *menu* acceptă meniuri de comenzi ce pot conţine maxim 7 opţiuni. Pentru a crea un obiect de tip *menu*, programul trebuie să specifice un tablou de opţiuni, un tablou de taste şi culorile de meniu, cum se arată în continuare:

```
option menu_a_options[7] = {{"Option One", 'A'},
                            {"Option Two", 'B'},
                            {"Option Three", 'C'}};

  menu a("Very Long Menu Title", menu_a_options, 37, 44);
```

În exemplul dat, culorile folosite vor fi text alb (37) pe fundal albastru (44). Aceste coduri de culoare sunt definite de driverul ANSI.

Observaţie: *Programul foloseşte secvenţele escape ANSI pentru a selecta culoarea ecranului şi a poziţiona cursorul. Dacă lucraţi într-un mediu PC, trebuie mai întâi să instalaţi driverul ANSI.SYS pentru ca sistemul să răspundă corect la secvenţele escape de culoare. Mai multe informaţii asupra folosirii acestor secvenţe în programele C++ le puteţi găsi în cartea „Jamsa's 1001 C/C++ Tips", Jamsa Press, 1993.*

Funcţia constructor a clasei *menu* determină linia şi coloana de plasare a opţiunilor de meniu, pentru ca acestea să fie centrate pe ecran în momentul afişării. În acest scop este nevoie de determinarea numărului de opţiuni din menu şi de lungimea textului fiecărei opţiuni.

Pentru a crea biblioteca clasei *menu*, plasaţi declaraţia clasei *menu* în fişierul MENU.CPP, cum se arată mai jos:

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>

struct option {
  char menu_text[64];
  char keystroke;
};

class menu {
  public:
    menu(char *title, option *menu_options,
```

```
      int foreground, int background);
    void show_menu(void);
    int get_option(void);
  private:
    char title[64];           // menu title
    option options[7];        // array of menu option strings
    int number_of_options;    // number of menu options
    int foreground;           // menu foreground color
    int background;           // menu background color
    int row;                  // row to start menu display
    int column;               // column to start menu display
    void set_colors(int foreground, int background);
    void set_row_and_column(int row, int column);
};

menu::menu(char *title, option *menu_options,
       int foreground, int background)
 {
   strcpy(menu::title, title);

   int number_of_options = 0;
   int longest_option = strlen(title);

   for (int i = 0; i < 7; i++)
    {
      int option_length = strlen(menu_options[i].menu_text);

      if (option_length > longest_option)
        longest_option = option_length;

      strcpy(menu::options[i].menu_text,
        menu_options[i].menu_text);
      menu::options[i].keystroke = menu_options[i].keystroke;
      if (options[i].menu_text)
        number_of_options++;
    }

   menu::number_of_options = number_of_options;
   menu::foreground = foreground;
   menu::background = background;

   column = (79 - longest_option) / 2;

   // Center options plus title, prompt and 2 blank lines
   row = (24 - (4 + number_of_options)) / 2;
 }

void menu::set_colors(int foreground, int background)
 {
   cout << "\033[" << foreground << ';' << background << 'm' <<
          endl;
```



```
   cout << "\033[2J";
 }

void menu::set_row_and_column(int row, int column)
 {
   cout << "\033[" << row << ';' << column << 'H';
 }

void menu::show_menu(void)
 {
   set_colors(foreground, background);
   set_row_and_column(row, column);

   cout << title << endl;

   for (int i = 0; i < number_of_options; i++)
     {
       set_row_and_column(row + 2 + i, column);
       cout << options[i].keystroke << ' ' <<
               options[i].menu_text;
     }

   set_row_and_column(row + i, column);
   cout << "Enter choice: ";
 }

int menu::get_option(void)
 {
   char option;
   int done = 0, choice;

   while (! done)
    {
      show_menu();
      option = cin.get();
      option = toupper(option);

      for (int i = 0; i < number_of_options; i++)
        if (option == options[i].keystroke)
          {
            choice = i;
            done = 1;
          }

      while (cin.get() != '\n')
        ;  // Eat characters up to and including \n
    }

   return (choice);
 }
```

Compilaţi apoi fişierul sursă pentru a crea un fişier obiect. Dacă lucraţi cu compilatorul Borland C++, folosiţi comanda:

```
C:\> BCC -c MENU.CPP <ENTER>
```

În caz că lucraţi cu compilatorul Microsoft Visual C++, folosiţi următoarea linie de comandă pentru compilarea fişierului:

```
C:\> CL -C MENU.CPP <ENTER>
```

Fişierul de bibliotecă se creează, în funcţie de compilator, folosind TLIB sau LIB:

```
C:\> TLIB MENU.LIB +MENU.OBJ <ENTER>

C:\> LIB MENU.LIB +MENU.OBJ; <ENTER>
```

Observaţie: *Nu uitaţi să includeţi simbolul ';' la sfârşitul comenzii LIB pentru a evita obligativitatea specificării numelor de fişiere.*

Următorul program, MENUDEMO.CPP, creează şi afişează diferite opţiuni de meniu folosind biblioteca clasei *menu*:

```
#include <iostream.h>
#include "menu.h"

void main(void)
{
   int choice;

   option menu_a_options[7] = {{"Option One", 'A'},
                               {"Option Two", 'B'},
                               {"Option Three", 'C'}};

   menu a("Very Long Menu Title", menu_a_options, 37, 44);

   choice = a.get_option();

   cout << endl << "You selected option " << choice+1 << endl;
   cout << "Press Enter to continue...";
   while (cin.get() != '\n')
     ;

   option menu_b_options[7] = {{"Perform Payroll", 'P'},
                               {"Update Accounts Payable", 'A'},
                               {"Print Checks", 'C'},
                               {"Quit", 'Q'}};

   menu b("Not So Quicken", menu_b_options, 37, 41);

   choice = b.get_option();

   cout << endl << "You selected option " << choice+1 << endl;
```



```
   cout << "Press Enter to continue...";
   while (cin.get() != '\n')
     ;

   // Restore white on black screen
   cout << "\033[37;40m\033[2J";
}
```

Pentru a compila şi lega acest program folosind Borland C++, daţi următoarea comandă:

```
C:\> BCC MENUDEMO.CPP MENU.LIB
```

Dacă lucraţi cu Microsoft Visual C++, folosiţi comanda:

```
C:\> CL MENUDEMO.CPP MENU.LIB
```

REZUMAT

O bibliotecă este o colecţie de funcţii precompilate folosite de programe pentru a realiza sarcini specifice. În cazul unei biblioteci de clase, funcţiile corespund unei anumite clase sau unei mulţimi de clase asociate. În acest capitol s-au examinat etapele ce trebuie efectuate pentru a crea o bibliotecă de clase, precum şi etapele necesare pentru a folosi o astfel de bibliotecă în programe. În Capitolul 18 veţi folosi aceste etape pentru a construi o bibliotecă de clase de liste dinamice. Înainte de a trece la Capitolul 18, asiguraţi-vă că aţi învăţat următoarele:

- ✓ O bibliotecă de funcţii obiect este un fişier ce conţine codul compilat al unui set de funcţii.
- ✓ Pentru a crea o bibliotecă obiect se scriu sau se asamblează funcţiile într-un fişier, se compilează fişierul, se foloseşte programul utilitar ce însoţeşte compilatorul pentru crearea fişierului de bibliotecă şi se construieşte un fişier antet ce conţine prototipurile funcţiilor.
- ✓ Pentru a folosi funcţiile de bibliotecă în program, se include fişierul antet corespunzător la începutul programului, se folosesc funcţiile de bibliotecă în codul programului sursă şi se leagă programul cu fişierul de bibliotecă.
- ✓ Pentru a crea o bibliotecă de clase se plasează declaraţia clasei şi a funcţiilor membru în fişierul sursă, se compilează fişierul sursă pentru a crea fişierul obiect, se foloseşte un program utilitar pentru a crea fişierul de bibliotecă şi se construieşte un fişier antet ce conţine declaraţia clasei.
- ✓ Pentru a folosi o bibliotecă de clase se include fişierul antet corespunzător la începutul programului, creează unul sau mai multe obiecte în program, apoi se compilează şi se leagă programul sursă la fişierul bibliotecă.

CAPITOLUL 18

CREAREA UNEI CLASE DE LISTE DINAMICE

Majoritatea programelor folosesc informaţii ce pot fi memorate şi prelucrate sub formă de liste dinamice. Aşa cum aţi citit pe scurt în Capitolul 7, listele dinamice sunt de preferat tablourilor deoarece folosesc eficient memoria disponibilă a calculatorului. Dacă aţi lucrat în trecut cu liste cu legături simple sau duble, ştiţi deja că astfel de liste sunt formate din *noduri*, ce conţin date, şi *pointeri*, ce conectează nodurile în listă. Datorită utilizării pe scară largă a listelor înlănţuite, programatorii încearcă întotdeauna modalităţi de a crea *liste generice* ce pot fi uşor adaptate pentru a satisface necesităţile diferitelor structuri de date folosite în programul curent sau chiar în alte programe.

Acest capitol vă ajută să creaţi o astfel de listă generică. Pentru a putea comuta lista de la o aplicaţie la alta, se creează şabloane de clasă pentru liste dinamice. În acest capitol se folosesc concepte prezentate în capitolele anterioare. Aveţi răbdare să exersaţi cu fiecare din programele prezentate. La început, sintaxa C++ pentru crearea listelor generice pare descurajantă. Totuşi, dacă citiţi comentariile ce însoţesc fiecare exemplu, veţi vedea că prelucrarea acestor liste nu este complicată. În momentul terminării acestui capitol, veţi cunoaşte:

- Ce este o listă cu legături simple
- Cum se foloseşte un şablon de clasă pentru a acepta variabile de diferite tipuri
- Ce este o listă cu legături simple
- Cum se parcurge o listă şi cum se contorizează apariţia unei anumite valori în listă
- Care este avantajul folosirii bibliotecilor de clase tip şabloane

ANALIZA UNEI LISTE CU LEGĂTURI SIMPLE

O *listă cu legături* este un grup de articole aranjate în secvenţă şi *legate* împreună, astfel încât fiecare membru sau nod să indice spre următorul articol din listă. Această structură este utilă îndeosebi pentru aplicaţii de tip listă, cum sunt bazele de date.

Înainte de a intra în detaliile unei liste generice, să analizăm un exemplu simplu pentru a ne obişnui cu listele înlănţuite. Pentru început, să creăm o listă cu legături simple ale cărei noduri să memoreze valorile de la 1 la 100.

Pentru a crea lista cu legături, programul foloseşte o clasă denumită *list* şi o structură denumită *node*. Clasa *list* conţine variabilele membru *first* şi *last* care indică începutul şi sfârşitul listei:

```
class list {
  public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
  private:
    node first;
    node *last;
};
```

După cum se poate constata, clasa conţine funcţiile membru *append_node* şi *show_list*. Funcţia *append_node* ataşează nodul specificat la sfârşitul listei înlănţuite. De asemenea, funcţia *show_list* parcurge lista, afişând valoarea din fiecare nod. Structura *node* conţine o valoare de tip *int* şi un pointer la următorul nod din listă:

```
struct node {
   int data;
   struct node *next;
};
```

Următorul program, SIMPLE.CPP, implementează lista simplă cu legături folosind clasa *list* şi structura *node*:

```
#include <iostream.h>

struct node {
  int value;
  struct node *next;
};

class list {
  public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
  private:
    node first;
    node *last;
};

void list::show_list(void)
```



```
 {
   node *current_node = first.next;

   while (current_node)
     {
       cout << current_node->value << endl;
       current_node = current_node->next;
     }
 }

void list::append_node(node *new_node)
 {
   last->next = new_node;
   last = new_node;
   last->next = NULL;
 }

void main(void)
 {
   list single;
   node *new_node;

   for (int i = 1; i <= 100; i++)
     {
       new_node = new node;
       new_node->value = i;
       single.append_node(new_node);
     }
   single.show_list();
 }
```

După cum se poate vedea, programul ataşează valorile la sfârşitul listei, folosind funcţia *append_node*. După ce toate valorile au fost atribuite listei, programul afişează conţinutul acesteia.



SĂ ÎNŢELEGEM FUNCŢIA APPEND_NODE

La prima vedere, funcţia *append_node* (de inserare la sfârşitul listei) poate fi derutantă, dar este foarte simplă. Reţineţi că variabila membru *last* a clasei *list* indică spre sfârşitul listei. De asemenea, variabila pointer *next* din structura *node* indică întotdeauna spre la următorul nod din listă.

Când introducem un nou nod în listă, acel nod trebuie să devină ultimul în listă. Aceasta se realizează atribuind noul nod variabilei membru *last*:

```
last = new_node;
```

Dar, aşa cum observaţi, aceasta nu este prima instrucţiune în funcţia *append_node*. Reamintiţi-vă că noul nod trebuie să devină ultimul nod din listă. Aceasta înseamnă că ultimul nod vechi va deveni penultimul nod în listă. Din acest motiv, vechiul nod ultim trebuie să indice noul nod ca fiind următorul în listă. Pentru aceasta trebuie să facem ca *next* să puncteze la *new_node* înainte de a-l face pe nodul *new_node* ultimul (*last*) în listă:

```
last->next = new_node;
```

În fiecare listă cu legături trebuie să cunoaştem când s-a ajuns la ultimul nod în listă. O practică de programare obişnuită este de a folosi valoarea NULL pentru a indica sfârşitul listei. Programul SIMPLE.CPP realizează acest lucru atribuind valoarea NULL pointerului *next* al noului nod, care între timp a devenit ultimul nod (*last*):

```
last->next = NULL;
```

Acum putem înţelege mai uşor funcţia constructor *list*:

NULL indică lipsa următorului nod — Primul şi ultimul nod coincid

```
list(void) { first.next = NULL; last = &first; };
```

Când se iniţializează lista, nodul *last* este acelaşi cu nodul *first*. De asemenea, pointerul *next* este NULL deoarece nu există următorul nod. Primul nod este primul, ultimul şi singurul nod în listă când aceasta este iniţializată de funcţia constructor.

Pentru a atribui valori listei cu legături, programul anterior a folosit următoarele instrucţiuni pentru alocarea unui nod şi atribuirea unei valori componentei nodului:

```
for (int i = 1; i <= 100; i++)
   {
     new_node = new node;
     new_node->value = i;
     single.append_node(new_node);
   }
```

Dacă se schimbă tipul *node* din structură în clasă, se poate reduce numărul de instrucţiuni necesare creării listei la următoarele:

```
for (int i = 1; i <= 100; i++)
    single.append_node(new node(i));
```

În acest caz, programul ataşează obiectul pointer iniţializat la sfârşitul listei cu legături într-un singur pas. Următorul program, NODECLAS.CPP, foloseşte clasa *node* pentru a construi lista cu legături:

```
#include <iostream.h>

class node {
```



```
  public:
    node(int value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node *next;
  private:
    int value;
};

class list {
  public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
  private:
    node first;
    node *last;
};

void list::show_list(void)
 {
   node *current_node = first.next;

   while (current_node)
     {
       current_node->show_value();
       current_node = current_node->next;
     }
 }

void list::append_node(node *new_node)
 {
   last->next = new_node;
   last = new_node;
   last->next = NULL;
 }

void main(void)
 {
   list single;

   for (int i = 1; i <= 100; i++)
     single.append_node(new node(i));

   single.show_list();
 }
```

După cum se observă, clasa *node* furnizează funcţii constructor şi o funcţie de afişare a valorilor din noduri.

FOLOSIREA UNUI ŞABLON DE CLASĂ

În programele precedente, lista cu legături conţinea numai valori de tip *int*. Folosind următorul şablon de clasă structura *node* poate fi modificată pentru a accepta valori de tip *int*, *float*, *long* ş.a.m.d.:

```
template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
  private:
    T value;
};
```

Următorul program, NEW_LIST.CPP, foloseşte şablonul de clasă *node* pentru a crea 3 liste, una pentru valori de tip *int*, una pentru valori de tip *float* şi alta pentru valori de tip *char*:

```
#include <iostream.h>

template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
  private:
    T value;
};

template<class T> class list {
  public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node<T> *new_node);
  private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
 {
   node<T> *current_node = first.next;

   while (current_node)
     {
       current_node->show_value();
       current_node = current_node->next;
```



```
     }
 }

template<class T> void list<T>::append_node(node<T> *new_node)
 {
   last->next = new_node;
   last = new_node;
   last->next = NULL;
 }

void main(void)
 {
   list<int> single;
   list<float> values;
   list<char> letters;

   for (int i = 1; i <= 10; i++)
     single.append_node(new node<int> (i));

   single.show_list();

   for (i = 0; i < 10; i++)
     values.append_node(new node<float> (0.1 * i));

   values.show_list();

   for (i = 'A'; i <= 'Z'; i++)
     letters.append_node(new node<char> (i));

   letters.show_list();
 }
```

Se poate observa că, folosind şablonul de clasă *node*, programul poate crea uşor liste cu legături capabile să memoreze date întregi, reale sau caractere.

Observaţie: *Dacă aveţi probleme în înţelegerea şabloanelor de clasă din programul NEW_LIST.CPP, recitiţi Capitolul 6. Singura diferenţă reală între programele NEW_LIST.CPP şi NODECLAS.CPP rezidă în funcţia* main. *Celelalte componente ale celor două programe sunt identice, excepţie făcând faptul că în NODECLAS.CPP clasele şi funcţiile sunt implementate prin şabloane. Încercaţi să analizaţi programul NEW_LIST.CPP fără a lua în considerare nici un aspect legat de şabloane.*

CÂND PROGRAMELE ACCEPTĂ LISTE CU LEGĂTURI DUBLE

Pentru inserţia şi ştergerea elementelor dintr-o listă cu legături, multe operaţii se pot simplifica folosind o *listă cu legături duble*. În sensul cel mai simplu, o listă cu legături duble conţine date şi doi pointeri, unul care indică nodul următor şi altul nodul precedent.

```
template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
    struct node<T> *previous;
  private:
    T value;
};
```

Să observăm că singura diferenţă între acest şablon şi şablonul *node* din NEW_LIST.CPP este apariţia pointerului *previous*:

```
struct node<T> *previous;
```

Următorul program, DOUBLY.CPP, modifică structura *node* şi funcţiile membru ale clasei *list* pentru a accepta o listă cu legături duble:

```
#include <iostream.h>

template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
    struct node<T> *previous;
  private:
    T value;
};

template<class T> class list {
  public:
    list(void) { first.next = NULL; last = &first;
      first.previous = NULL; };
    void show_list(void);
    void append_node(node<T> *new_node);
    void show_reverse(void);
  private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
 {
   node<T> *current_node = first.next;

   while (current_node)
     {
       current_node->show_value();
```



```
       current_node = current_node->next;
     }
 }

template<class T> void list<T>::show_reverse(void)
 {
   node<T> *current_node = last;

   while (current_node->previous)
     {
       current_node->show_value();
       current_node = current_node->previous;
     }
 }

template<class T> void list<T>::append_node(node<T> *new_node)
 {
   last->next = new_node;
   new_node->previous = last;
   last = new_node;
   last->next = NULL;
 }

void main(void)
 {
   list<int> single;
   list<float> values;
   list<char> letters;

   for (int i = 1; i <= 10; i++)
     single.append_node(new node<int> (i));

   single.show_list();

   for (i = 0; i < 10; i++)
     values.append_node(new node<float> (0.1 * i));

   values.show_list();

   for (i = 'A'; i <= 'Z'; i++)
     letters.append_node(new node<char> (i));

   letters.show_list();

   letters.show_reverse();
 }
```

După cum se observă, clasa adaugă funcţia *show_reverse* care afişează conţinutul listei în ordine inversă. Deoarece fiecare nod conţine acum un pointer la nodul precedent, parcurgerea listei în ordine inversă este foarte simplă.

Funcţia *append_node* este similară celei din programele anterioare, cu excepţia faptului că apare pointerul *previous* ce indică nodul precedent din listă.

De asemenea, funcţia constructor *list* trebuie să iniţializeze atât *previous* cât şi *next* la valoarea NULL. Ca şi în cazul unei liste cu legături simple când este iniţializată de funcţia constructor *list*, primul nod al unei liste cu legături duble este, în momentul iniţializării, primul, ultimul şi singurul nod în listă - nu există noduri următoare (*next*) sau precedente (*previous*).

Fiecare din programele precedente au adăugat elemente la lista cu legături prin simpla ataşare a acestora la finalul listei. Următorul program, INSERT.CPP, include în clasa *list* şi funcţia membru *insert_node* ce permite inserţia valorilor în listă în ordine crescătoare. Folosind o listă cu legături duble, programul poate insera cu uşurinţă un element în listă:

```
#include <iostream.h>
#include <stdlib.h>

template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    T get_value(void) { return(value); };
    struct node<T> *next;
    struct node<T> *previous;
  private:
    T value;
};

template<class T> class list {
  public:
    list(void) { first.next = NULL; last = &first;
      first.previous = NULL; };
    void show_list(void);
    void append_node(node<T> *new_node);
    void show_reverse(void);
    void insert_node(node<T> *new_node);
  private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
 {
   node<T> *current_node = first.next;

   while (current_node)
     {
       current_node->show_value();
```



```
       current_node = current_node->next;
     }
 }

template<class T> void list<T>::show_reverse(void)
 {
   node<T> *current_node = last;

   while (current_node->previous)
     {
       current_node->show_value();
       current_node = current_node->previous;
     }
 }

template<class T> void list<T>::append_node(node<T> *new_node)
 {
   last->next = new_node;
   new_node->previous = last;
   last = new_node;
   last->next = NULL;
 }

template<class T> void list<T>::insert_node(node<T> *new_node)
 {
   node<T> *current_node = first.next;
   node<T> *previous_node = &first;

   while ((current_node) &&
       (current_node->get_value() < new_node->get_value()))
     {
       current_node = current_node->next;
       previous_node = previous_node->next;
     }

   previous_node->next = new_node;
   new_node->previous = previous_node;
   new_node->next = current_node;
 }

void main(void)
 {
   list<float> values;

   for (int i = 0; i < 10; i++)
     values.insert_node(new node<float> (rand()));

   values.show_list();
 }
```

Acţiunea funcţiei *insert_node* este evidentă. Instrucţiunea *while* determină parcurgerea listei nod cu nod şi compară cu valoarea din *new_node*:

```
while ((current_node) &&
      (current_node->get_value() < new_node->get_value()))
    {
      current_node = current_node->next;
      previous_node = previous_node->next;
    }
```

Când valoarea din *current_node* este mai mică decât valoarea din *new_node*, ciclul *while* se încheie. Noul nod este inserat cu următoarele instrucţiuni:

```
previous_node->next = new_node;
   new_node->previous = previous_node;
   new_node->next = current_node;
```

Dacă toate valorile din nod sunt mai mici decât valoarea din *new_node* atunci funcţia *insert_node* ataşează noul nod în finalul listei.

ADĂUGAREA UNOR MEMBRI AI CLASEI

Programele anterioare au realizat operaţii simple cu liste, ca inserţia sau ataşarea unui nod sau afişarea conţinutului nodurilor. În multe situaţii este necesară căutarea unei anumite valori într-o listă cu legături sau efectuarea unei anumite operaţii cu valorile din noduri. Următorul program, SEARCH.CPP, de exemplu, foloseşte funcţia membru *search* pentru a căuta într-o listă cu legături prima apariţie a unei valori specificate. În plus, programul foloseşte funcţia membru *cout* pentru a contoriza numărul de apariţii ale valorii specificate:

```
#include <iostream.h>
#include <stdlib.h>

template<class T> class node {
  public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    T get_value(void) { return(value); };
    struct node<T> *next;
    struct node<T> *previous;
  private:
    T value;
};

template<class T> class list {
  public:
    list(void) { first.next = NULL; last = &first;
      first.previous = NULL; };
    void show_list(void);
    void append_node(node<T> *new_node);
    void show_reverse(void);
    void insert_node(node<T> *new_node);
    node<T> *search(T value);
    int count(T value);
  private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
 {
   node<T> *current_node = first.next;

   while (current_node)
     {
       current_node->show_value();
       current_node = current_node->next;
     }
 }

template<class T> void list<T>::show_reverse(void)
 {
   node<T> *current_node = last;

   while (current_node->previous)
     {
       current_node->show_value();
       current_node = current_node->previous;
     }
 }

template<class T> void list<T>::append_node(node<T> *new_node)
 {
   last->next = new_node;
   new_node->previous = last;
   last = new_node;
   last->next = NULL;
 }

template<class T> void list<T>::insert_node(node<T> *new_node)
 {
   node<T> *current_node = first.next;
   node<T> *previous_node = &first;

   while ((current_node) &&
       (current_node->get_value() < new_node->get_value()))
     {
       current_node = current_node->next;
```

```
        previous_node = previous_node->next;
      }

    previous_node->next = new_node;
    new_node->previous = previous_node;
    new_node->next = current_node;
  }

template<class T> node<T> *list<T>::search(T value)
 {
   node<T> *current_node = first.next;

   while (current_node)
     if (current_node->get_value() == value)
       return(current_node);
     else
       current_node = current_node->next;

   return(NULL);
 }

template<class T> int list<T>::count(T value)
 {
   node<T> *current_node = first.next;

   int count = 0;

   while (current_node)
    {
      if (current_node->get_value() == value)
        count++;
      current_node = current_node->next;
    }
   return(count);
 }

void main(void)
 {
   list<float> values;

   for (int i = 0; i < 10; i++)
     values.insert_node(new node<float> (2.5 * i));

   values.show_list();

   if (values.search(7.5))
     cout << "Found the value 7.5" << endl;
   else
     cout << "The value 7.5 was not found" << endl;
```



```
   cout << "7.5 was found " << values.count(7.5) << " times" <<
           endl;
 }
```

După cum se observă, programul foloseşte funcţia *search* pentru a căuta în listă valoarea 7.5. De asemenea, programul foloseşte funcţia *count* pentru a număra apariţiile valorii 7.5 în listă.

Într-o manieră similară funcţiei *insert_node* din programul precedent, funcţia *search* parcurge lista astfel:

```
while (current_node)
     if (current_node->get_value() == value)
       return(current_node);
     else
       current_node = current_node->next;

   return(NULL);
```

Acum însă, în loc de a căuta o valoare mai mică decât parametrul funcţiei, cum se proceda în *insert_node*, se caută o valoare echivalentă *current_node*. Dacă este găsită o valoare echivalentă, funcţia *search* va returna imediat valoarea nodului:

```
if (current_node->get_value() == value)
      return(current_node);
```

Dacă valoarea nu este găsită, funcţia *search* returnează NULL.

Funcţia *count* este aproape identică cu *search*. Dar, în loc de a reveni în program imediat ce este găsită valoarea, funcţia *count* incrementează o variabilă contor de tip *int*:

```
while (current_node)
    {
      if (current_node->get_value() == value)
        count++; ———————————————————— Incrementarea variabilei contor
      current_node = current_node->next;
    }
   return(count);
```

Spre deosebire de celelalte funcţii membru examinate, funcţia *count* traversează întotdeauna întreaga listă înainte de a reveni în program. Valoarea variabilei contor, care poate fi şi zero, este întotdeauna returnată de *count*.

ŞABLOANE ŞI BIBLIOTECI DE CLASE

În Capitolul 17 aţi învăţat cum să creaţi o bibliotecă de clase. Programele prezentate în acest capitol au folosit şabloane pentru a crea liste generice. Aşa cum aţi învăţat în Capitolul 6, în timpul compilării, compilatorul C++ creează clasele

și funcțiile de clasă corecte, pe baza tipurilor specificate la declararea obiectelor. Din păcate, deoarece șabloanele sunt folosite de compilator, este dificil de a crea biblioteci de clase bazate pe șabloane. În cazul programelor anterioare, ar fi mai comodă și mai flexibilă plasarea șabloanelor și funcțiilor într-un fișier antet, decât încercarea de a construi o bibliotecă de clase.

REZUMAT

Acest capitol a examinat posibilitățile de folosire a șabloanelor pentru a crea o clasă de liste generice cu legături. Programele prezentate în acest capitol ilustrează multe din conceptele discutate în această carte. Este bine să exersați cu fiecare din aceste programe. Înainte de a vă continua aventura cu C++, asigurați-vă că ați învățat următoarele:

- ✓ Într-o listă cu legături simple, fiecare nod indică spre nodul următor, iar ultimul nod indică NULL.
- ✓ Instrucțiunea *list <type>* într-un șablon de clasă determină acceptarea unor variabile de diferite tipuri, indicate de *type*.
- ✓ Într-o listă cu legături duble, fiecare nod are un pointer la următorul nod și un pointer la nodul precedent.
- ✓ Într-un șablon de clasă se pot introduce funcțiile membru *search* și *count* pentru a căuta și număra aparițiile unei anumite valori.
- ✓ Este foarte dificil de a crea biblioteci de șabloane de clase. Introducerea șabloanelor și a funcțiilor într-un fișier antet este mai ușoară și mai flexibilă decât încercarea de a crea o bibliotecă de șabloane de clase.

Index

D

E

F

Index

G

H

I

L

M

N

O

P

R

S

Ş

T

U

V

W

Z